Lock-free parallel and concurrent garbage collection by mark&sweep

Gao, H.; Groote, J. F.; Hesselink, W. H.

Published in:
Science of computer programming

DOI:
10.1016/j.scico.2006.10.001

Publication date:
2007

Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 64 (2007) 341–374
Science of Computer Programming
www.elsevier.com/locate/scico

Lock-free parallel and concurrent garbage collection by mark&sweep

H. Gao[a], J.F. Groote[b], W.H. Hesselink[c,*]

[a] *School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 610054, China*
[b] *Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands*
[c] *Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands*

Received 22 March 2006; received in revised form 29 September 2006; accepted 6 October 2006
Available online 20 November 2006

Abstract

This paper presents a lock-free algorithm for mark&sweep garbage collection (GC) in a realistic model using synchronization primitives *load-linked/store-conditional* (*LL*/*SC*) or *compare-and-swap* (*CAS*) offered by machine architectures. The algorithm is concurrent in the sense that garbage collection can run concurrently with the application (the *mutator* threads). It is parallel in that garbage collection itself may employ several concurrent *collector* threads.

We first design and prove an algorithm with a coarse grain of atomicity and subsequently apply the reduction method developed and verified in [H. Gao, W.H. Hesselink, A formal reduction for lock-free parallel algorithms, in: Proceedings of the 16th Conference on Computer Aided Verification, CAV, July 2004] to implement the high-level atomic steps by means of the low-level primitives. Even the correctness of the coarse grain algorithm is very delicate due to the presence of concurrent mutator and collector threads. We have verified it therefore by means of the interactive theorem prover PVS.

Keywords: Garbage collection; Lock-free; Shared-memory; Correctness; Theorem proving

1. Introduction

On shared-memory multiprocessors, processes coordinate with each other via shared data structures. To ensure the consistency of these concurrent objects, processes need a mechanism for synchronizing their access. In such a system the programmer typically has to synchronize access to shared data by different processes explicitly to ensure correct behavior of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. In fact, the operations of different processes on a shared data structure should appear to be serialized[1] so that the object state is kept coherent after each operation.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. A *lock-free* (also called *non-blocking*) implementation of a shared object guarantees

* Corresponding address: University of Groningen, Informatica, Blauwborgje 10 Postbus 800, 9700 AVGroningen, The Netherlands. Tel.: +31 503633933; fax: +31 503633800.
E-mail addresses: huigao@uestc.edu.cn (H. Gao), jfg@win.tue.nl (J.F. Groote), w.h.hesselink@rug.nl (W.H. Hesselink).

[1] If two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2006.10.001

that it is always the case that within a finite number of steps some process trying to perform an operation on the object will complete its task, independent of the activity and speed of other processes [25]. As lock-free algorithms are built without locks, they are immune from the aforementioned problems. In addition, lock-free algorithms can offer progress guarantees. A number of researchers [4,6,25,26,40,42] have proposed techniques for designing lock-free implementations. Essential for such implementations are advanced machine instructions such as *load-linked/store-conditional* (*LL*/*SC*), or *compare-and-swap* (*CAS*).

In this paper we propose a lock-free implementation of mark&sweep *garbage collection* (GC). Garbage *collector*s are employed to identify at run-time which objects are no longer referenced by the *mutator*s (i.e., user programs that use and modify the objects). The heap space occupied by these objects is said to be *garbage* and must be recycled for subsequent new objects. The garbage collectors reclaim all garbage by adding them to a so called *free-list*, which keeps track of free memory. Some programming languages (e.g., C, C++) force or allow the programs to do their own memory management, which means that programs are required to delete objects that they allocate in memory. However, this task is so difficult that nontrivial applications often exhibit incorrect behavior as the result of memory leaks or dangling pointers. To relieve programmers of many memory-management problems, it is preferable to offer GC that is triggered during memory allocation when the amount of free memory falls below some threshold or after a certain number of allocations.

There are several basic strategies for GC: reference counting, e.g., [12,35,39,47], mark&sweep, e.g., [2,5,13–15] and copying, e.g., [27,30–32,48]. Reference counting algorithms can do their job incrementally (the entire heap need not be collected at once, resulting in shorter collection pauses), but impose overhead on the mutators and fail to reclaim circular garbage. Mark&sweep algorithms can reclaim circular structures, and don't place any burden on the mutators like reference counting algorithms do, but tend to leave the heap fragmented. Copying algorithms can reduce fragmentation, but add the cost of copying data from one space to another and require twice as much memory as a mark&sweep collector. For a more detailed introduction to garbage collection and memory management the reader is referred to [35].

One often encounters GC algorithms (e.g., [7,15,16,52]) that employ "stop-the-world" mechanisms, which suspend all normal running threads and then perform GC. Such an algorithm introduces a global synchronization point between all threads and tends to become a scaling bottleneck that limits processor utilization. In particular, a "stop-the-world" mechanism violates non-blockingness. This is unacceptable when the system must guarantee response time of interactive applications. Therefore, to achieve parallel speed-ups on shared-memory multiprocessors, lock-free algorithms are of interest [25,27,37,54,55].

There are several lock-free GC algorithms in the literature. The first one is due to Herlihy and Moss [27]. They present a lock-free copying GC algorithm, which uses copying for moving objects to avoid blocking synchronization. In their algorithm, the failure of a participating thread can indefinitely prevent the freeing of unbounded memory. In [30], Hesselink and Groote give a wait-free (wait-freedom is stronger than lock-freedom) GC algorithm using reference counting. However, this collector applies only to a restricted programming model, in which objects are not allowed to be modified between creation and deletion, and is therefore generally limited. Detlefs et al. [12] provide a lock-free GC algorithm using reference counting. The approach relies on a strong hardware primitive, namely *double-compare-and-swap* (*DCAS*) for atomic update of two completely distinct words in memory. Michael [43] presents an efficient lock-free memory management algorithm that does not require special operating system or hardware support. However, his algorithm only guarantees an upper bound on the number of removed nodes not yet freed at any time. This is undesirable because a single garbage node might induce a large amount of occupied resources and might never be reclaimed. See also [26].

Our lock-free mark&sweep algorithm is nonintrusive and features high performance and reliability. We make no assumption on the maximum numbers of mutators and collectors that can operate concurrently. Our mutators are allowed to add nodes and links to the memory graph, and to inspect and modify data in the nodes, but not to delete or modify links. Instead, by selecting root nodes, they specify the accessible part of the memory graph, and therewith the part that holds garbage. The precise interface is described and discussed in Section 2. The performance of GC can be improved when more processors are involved in it.

The correctness properties of concurrent algorithms are seldom easy to verify. This is in general even harder for lock-free algorithms. Our previous work [17,18] shows that providing correctness proofs for such algorithms requires huge amounts of effort, time, and skill. In [21,22], we developed two reduction theorems that enable us to reason about a lock-free program to be designed on a higher level than the synchronization primitives *LL*/*SC* and *CAS*. The

```
Constant
    P = number of mutators;
    N = number of nodes;
    C = upper bound of number of children;
Type
    Index = [1 ... N];
    Process = [1 ... P];
    nodeType : record =
        application data;
        arity : [1 ... C];                      % number of children
        child : array [1 ... C] of Index;       % pointers to children
    end
Shared variables
    Node : array [Index] of nodeType;           % N shared nodes
    Mbox : array [Process, Process] of Index ∪ {0};   % mailboxes
    free : subset of Index;                     % the set of free nodes
Private mutator variables
    roots : subset of Index;                    % a set of root nodes
```

Fig. 1. Specification data structure.

reduction theorems are based on refinement mappings as described by Lamport [38], which are used to prove that a low-level specification correctly implements a high-level one. Using the reduction theorems, fewer invariants are required and some invariants are easier to discover and formulate since one needs not go into the internal structure of the final implementation. In particular, the nested loops in the low-level algorithm reduce to single loops in the high-level algorithm.

We used the higher-order interactive theorem prover PVS [51] for the verifications of the high-level algorithm and of the reduction theorems. It is worth noting that there are not many computer-checked correctness proofs of concurrent GC algorithms. Versions of the GC algorithms of [13,5] have been verified in [53,33,24] with the theorem provers NQTHM and PVS. These algorithms contain a single garbage collector. In [47], Moreau and Duprat model a distributed reference counting algorithm and prove safety and liveness properties with Coq [11].

This article is a minor revision of [19]. A shorter version has been published as [20].

Overview of the paper

Section 2 contains the specification of the garbage collector and the interface offered to the users. The high-level implementation is presented in Section 3. In Section 4, the correctness properties are proven. The proof is based on a list of invariants and lemmas, presented in Appendix A, while the relationships between the invariants are given by a dependency graph in Appendix B. Section 5 describes the transformation rules to implement the coarse grain atomicity of the high-level algorithm by means of the low-level primitives LL and SC via the reduction theorem of [21]. The result is given in Appendix C. In Section 6, we present some experimental data about the implementation. Conclusions are drawn in Section 7.

2. Specification

The data structure for the specification is summarized in Fig. 1. We assume a fixed set of nodes, each of which is identified by a unique index between 1 and N for some $N \in \mathbb{N}$. To specify garbage collection, we introduce a specification variable `free` to hold the set of indices available for allocation of new objects by the application processes. The set `free` is filled by the garbage collectors.

The indices outside `free` form a finite directed graph of varying structure, called the heap, see Fig. 2. Each node in the graph points to zero or more children, and the descendent relation may be circular. The number of children of a node x is given by `arity`$[x]$, which is used as an alias of `Node`$[x]$.`arity`. We let C be the upper bound of the arities, which may be set by the implementator arbitrarily. We use `child`$[x, j]$ as an alias of `Node`$[x]$.`child`$[j]$. This is the pointer to the jth child of x, where $1 \leq j \leq$ `arity`$[x] \leq C$.

The application processes that inspect and modify the graph are traditionally called *mutators*. A node is called a *root* when some mutator has direct read access to it (such as global/static variables, stack locations and registers of

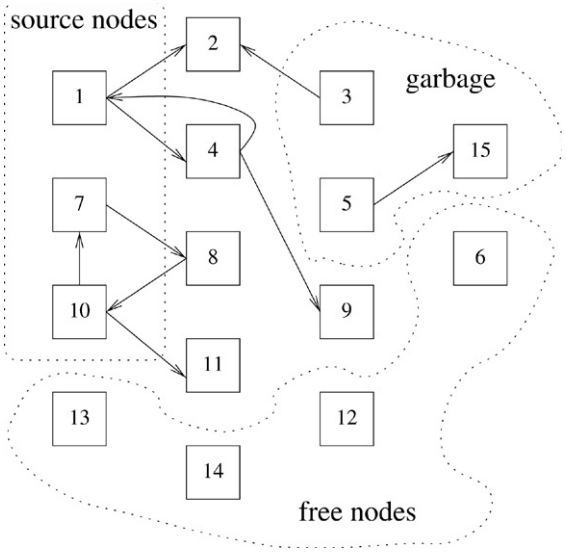

Fig. 2. A graph representation of the memory.

the system). Each mutator p maintains a private set $roots_p$ that holds its root nodes. The set $Roots$ is the union of all $roots_p$ for all mutators p.

Access to nodes can be transferred between mutators. We assume that there is a two-dimensional array `Mbox` indexed with a pair of mutators that serves as mailboxes. If mutator p allows mutator q to access some node x, it writes x at $\texttt{Mbox}[p, q]$ using $Send$. Mutator q then obtains access to node x by calling $Receive$. Notice that mutators can safely share many nodes by keeping them in their set *roots*, the mailboxes only serve to share or transfer access rights.

We call a node a *source node* if the node is either in $Roots$ or in some mailbox. A node is called *accessible* iff it is reachable by following a chain of pointers from a source node. Free nodes must not be accessible. Only nodes in the `free` set are allowed to be allocated by the mutators. A node is said to be a *garbage node* if it is neither accessible nor in the `free` set. We thus have that free, accessible and garbage partition the set of nodes. It is the aim of garbage collection to reclaim all garbage nodes by placing them into the `free` set.

To formalize accessibility, we define

$$R(p,\ x)\ \equiv\ (\exists z \in roots_p : z \xrightarrow{*} x),$$
$$R(x)\ \equiv\ (\exists z \in Roots : z \xrightarrow{*} x) \vee (\exists p,\ q \in \texttt{Process} : \texttt{Mbox}[p,\ q] \xrightarrow{*} x),$$

where the reachability relation $\xrightarrow{*}$ is the reflexive transitive closure of the relation $\rightarrow$ on nodes defined by

$$z \rightarrow x\ \equiv\ (\exists k \in [1 \ldots \texttt{arity}[z]] : \texttt{child}[z,\ k] = x).$$

According to the definitions, a node x is accessible iff $R(x)$ holds. Process p is said to have access to node x if $R(p, x)$ holds. Obviously, $R(p, x)$ implies $R(x)$. The fact that a node x is a garbage node is formalized by:

$$\texttt{garbage}(x)\ \equiv\ \neg R(x) \wedge x \notin \texttt{free}.$$

GC does not modify the memory graph (children or arities of nodes) but only repeatedly adds garbage nodes to the `free` set by executing:

proc *GCollect*()
⟨ **choose** $x \in \texttt{Index}$ **such that** $\texttt{garbage}(x)$; $\texttt{free} := \texttt{free} \cup \{x\}$; ⟩

Here, and henceforth, we use angular brackets ⟨ ⟩ to indicate that embraced statements are (thought to be) executed atomically.

To specify that GC does happen and is eventually exhaustive, we give the progress property of the collectors specified as follows:

$$\texttt{garbage}(x) \;\Rightarrow\; \Diamond(x \in \texttt{free})$$

that is, every garbage node will be eventually put into the `free` set by a garbage collector.

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space. There can be several processes running on a single processor. We assume that there are P concurrently executing mutators. In the text of a procedure, we use *self* to indicate the process identifier of the process that invokes the procedure. The interface consists of a shared data structure of nodes, and a number of procedures that can be called by the mutators.

We provide procedures that can read and modify the reachable part of the memory graph (from source nodes). An application programmer can assume that the behavior of the routines to access the data is as provided here. In this sense these routines are the specification of our algorithm. In the next section we provide implementable routines with the same behavior as specified here. The specification procedures are *Create*, *AddChild*, *GetChild*, *Make*, *Protect*, *UnProtect*, *Send*, *Receive* and *Check*. We use braces { } to indicate a precondition that must hold when invoking a certain procedure.

proc *Create*() : `Index`
  local x : `Index`;
  ⟨ **when** `available` **extract** x **from** `free`;
    `arity`$[x] := 0$; $roots_{self} := roots_{self} \cup \{x\}$; ⟩
  return x;

Create makes a new node without any children and returns its index. It may have to wait for nodes to become available, and it may trigger garbage collection.

proc *AddChild*(x, y : `Index`) : `Bool`
{ $R(self, x) \wedge R(self, y)$ }
  local suc : `Bool`;
  ⟨ $suc := (\texttt{arity}[x] < C)$;
    if suc **then** `arity`$[x]$++; `child`$[x,$ `arity`$[x]] := y$; **fi** ⟩
  return suc;

Recall that C is the maximal arity. If possible, *AddChild* adds y as a new child of x. It returns whether it succeeded.

proc *Make*(c : **array** [ ] **of** `Index`, $n : 1 \ldots C$) : `Index`
{ $\forall j \in [1 \ldots n] : R(self, c[j])$ }
  local x : `Index`; $j : \mathbb{N}$;
  ⟨ **when** `available` **extract** x **from** `free`;
    for $j := 1$ **to** n **do** `child`$[x, j] := c[j]$ **od**;
    `arity`$[x] := n$; $roots_{self} := roots_{self} \cup \{x\}$; ⟩
  return x;

Make is an atomic combination of *Create* and *AddChild* to create a node with a given sequence of children.

proc *GetChild*(x : `Index`, $rth : \mathbb{N}$) : `Index` $\cup \{0\}$
{ $R(self, x)$ }
  local y : `Index` $\cup \{0\}$;
  ⟨ **if** $1 \le rth \le$ `arity`$[x]$ **then** $y :=$ `child`$[x, rth]$; **else** $y := 0$; **fi** ⟩
  return y;

If possible, *GetChild* returns the index of the rth child of x. Otherwise it returns 0.

proc *Protect*(x : `Index`)
{ $R(self, x) \wedge x \notin roots_{self}$ }
  ⟨ $roots_{self} := roots_{self} \cup \{x\}$; ⟩
  return;

A mutator uses *Protect* to declare its interest in this node and its descendants.

```
proc UnProtect(z : Index)
{ z ∈ roots_self }
    ⟨ roots_self := roots_self \ {z}; ⟩
    return;
```

A mutator uses *Unprotect* when it needs the node no longer.

```
proc Send(x : Index, r : Process)
{ R(self, x) ∧ Mbox[self, r] = 0 }
    ⟨ Mbox[self, r] := x; ⟩
    return;
```

A mutator, say p, can *Send* a node x to another mutator, say r, when the channel $\texttt{Mbox}[p, r]$ is empty. When the channel is full, node r can *Receive* the node.

```
proc Receive(r : Process) : Index
{ Mbox[r, self] ≠ 0 }
    local x : Index;
    ⟨ x := Mbox[r, self];
      Mbox[r, self] := 0; roots_self := roots_self ∪ {x}; ⟩
    return x;
```

Check serves to test whether the channel is empty or full.

```
proc Check(r, q : Process) : Bool
    local suc : Bool;
    ⟨ suc := (Mbox[r, q] = 0); ⟩
    return suc;
```

The application programmers are responsible for ensuring that an offered procedure is called only when its precondition (enclosed by braces if there is any) holds. It is a proof obligation for us that all preconditions of any interface procedure are stable from the perspective of the calling mutator.

A mutator may continuously allocate a node, add some pointers in the memory, and remove a node from its *roots* set or mailbox. When an allocation request is made, the mutator tries to find a free node (see procedures *Create* and *Make*). This is in line with modern memory management systems that allow sharing of common subparts (see e.g. the ATerm library [8]). The condition "available" in *Create* and *Make* is implementation dependent. When an allocation request cannot be met from the free memory, the mutator either waits, or invokes a new round of GC to free more garbage, or expands the current heap by requesting more memory from the operating system. The threshold value that determines whether or not to invoke a new round of GC can be customized by the user.

The interface is designed in such a way that, when $R(p, x)$ holds, no mutator other than p can falsify $R(p, x)$. This means that every mutator can justify the accessibility of node x by checking $R(p, x)$ (via repeatedly reading arities and children of nodes) without worrying about possible interference from other mutators. Indeed, no mutator is able to decrease $\texttt{arity}[x]$ or modify $\texttt{child}[x, j]$ for $1 \leq j \leq \texttt{arity}[x]$ when node x is accessible (see procedures *AddChild* and *Make*).

Instead, the interface only allows extension of the graph by making new nodes and by adding already accessible children. This restriction is stronger than elsewhere, e.g., [5,36]. Yet, it can be justified as follows. In some systems, like the ATerm library of [8], subterms (i.e., children) are never changed since this would conflict with sharing of subnodes, which is essential for effective use of memory. In such a case, if a child needs to be changed, data need to be copied. For such classes of systems the interface is clearly very suitable. It just requires a style that is often associated with functional programming. On the other hand, even without garbage collection, systems in which mutators can modify the graph concurrently are very difficult to handle. Of course, we would be very interested in extending our results to the case where subnodes can be changed and removed. This however is more difficult and we leave this as an open question.

The intention of *UnProtect* is that it makes the node and its descendants eligible for garbage collection unless some other mutator wants to keep them. Via *Send*, *Receive* and *Check*, our algorithm can be used in a distributed system, in which all processors cooperatively traverse the entire data graph by exchanging "messages" to access remote nodes.

3. The high-level implementation

The idea behind mark&sweep GC algorithms in use is to first recursively trace all reachable nodes starting from root nodes, then nodes not reached are considered garbage and can be collected. We present a lock-free implementation that comes close to the classical mark&sweep algorithms. Since we allow access to nodes to be transferred between mutators via mailboxes, we have strengthened the definition of garbage to non-reachability from source nodes instead of from root nodes.

We refer to [1], chapter 6, for the semantics of concurrent algorithms with shared variables. The problem with concurrent algorithms is that different concurrently executing processes can interfere with each other by changing shared variables. The processes themselves are sequential programs, but the meaning of a system of processes is defined by means of all possible interleavings of their atomic commands.

For the sake of simplicity, of both presentation and proof, we first implement the specification at a rather coarse grain of atomicity. To restrict the points of interference, we use so-called atomic regions whose execution cannot be interrupted by other processes. A compound command S is declared to be an atomic region by enclosing it as $\langle S \rangle$. In order to transform this high-level algorithm into a low-level algorithm by means of the reduction theorem of [21] or [22], we ensure that every atomic region of the high-level algorithm refers to at most one shared node.

Notational conventions

Recall that there are P mutators with process identifiers ranging from 1 up to P and N nodes labeled by indices from 1 up to N. The mutators can become temporary collectors, and there may be other collector processes with their own process identifiers.

Unless otherwise specified, we assume that the free variables p, q and r range over process identifiers and the free variables w, x, y and z range over node indices. Since the same sequential program can be executed by all processes, we adopt the convention that, when we discuss a private variable of a particular process, it is subscripted by the process identifier. In particular, pc_p is the program location of process p. We use $\mathbb{N}$ to denote the set of natural numbers, starting at 0. If S is a finite set, we write $\sharp(S)$ to denote its number of elements.

3.1. Data structure

The data structure of the high-level implementation is shown in Fig. 3 as an extension of Fig. 1. We redefine the type `nodeType` here to hold additional information that only serves in the GC algorithm. The application data are omitted since they are irrelevant for GC. For every field f of `nodeType`, we use $f[x]$ as an alias of $\texttt{Node}[x].f$ (we did this for `arity` in Section 2).

Besides fields `arity` and `child`, each node has one of three colors: *white*, *black* and *grey*, which is stored in the field `color`. The *white* nodes are free, i.e., the specification variable `free` is now defined as the set of the *white* nodes. When there are no processes collecting garbage, all other nodes are *black*. In the first phase of GC, all *black* nodes will be painted *grey*. In the second phase of GC, all reachable *grey* nodes will be painted *black* again. In the third phase of GC, the remaining *grey* nodes will be painted *white*. Such color-coding of garbage collecting stages goes back to [13].

Since any accessible node must not be freed as garbage, the system needs to keep track of source nodes that have been created by a mutator and may still be referred to by other mutators. For safety, a process is not allowed to inspect another process's private variable such as *roots*. Instead, we introduce a field `srcnt` for each node to count all references (roots and mailboxes) to the node as a source node. Intentionally, we would like to have something like[2]:

$$\texttt{srcnt}[x] = \sharp(\{p \mid x \in \mathit{roots}_p\}) + \sharp(\{(p,\ q) \mid \texttt{Mbox}[p,\ q] = x\}).$$

Therefore, each collector can recognize a source node by checking if its `srcnt` field is positive. We define:

[2] The precise formula is invariant *I5* in Appendix A.

Type
`colorType` = {*white, black, grey*};
`nodeType` : **record** =
`arity` : $\mathbb{N}$; % number of children
`child` : **array** [1 ... C] **of** `Index`; % pointers to children
`color` : `colorType`; % holds the color of the node
`srcnt` : $\mathbb{N}$; % reference counter for a source node
`freecnt` : $\mathbb{N}$; % dereference counter for a source node
`ari` : $\mathbb{N}$; % number of children at the beginning of GC
`father` : $\mathbb{N} \cup \{-1\}$; % records the parent node GC traverses
`round` : $\mathbb{N}$; % the latest round of GC involved in, starting from 1
end
Shared variables
`shRnd` : $\mathbb{N}$; % the version of the current round of GC
Private collector variables
rnd : $\mathbb{N}$; % private copy of "shRnd", initially 0
toBeC : **subset of** `Index`; % a set of nodes to be checked

Initialization:
`shRnd` $= 1 \wedge \forall x \in$ `Index` : `round`$[x] = 1$;
all other variables are equal to the minimal values in their respective domains.

Fig. 3. Additional data structure of the implementation.

$$R1(x) \equiv (\exists z : \texttt{srcnt}[z] > 0 \wedge z \xrightarrow{*} x),$$

and we have $R(x) \Rightarrow R1(x)$. We do not apply other reference counting to the nodes, since manipulating reference counters is slow and may incur expensive overhead with every duplication and deletion of the pointers.

The main difficulty with tracing the memory graph is that the memory structure can change during GC (root nodes can be added or removed, mailboxes can change, children can be added). In order to solve this problem, we need some coordination between mutators and collectors to take the view of the memory graph, on which all collectors work. To avoid possible interference between mutators and collectors (we will explain this later), the update of the field `srcnt` of the node in *UnProtect*, upon deletion from the *roots* set, is postponed until the end of GC. We use the field `freecnt` to count the postponed decrementings of `srcnt`. Field `ari`$[x]$ contains the number of children node x has at the beginning of GC. Field `father`$[x]$ holds the parent node of x in the tree traversed from a source node by the collectors.

Since there may be several concurrent collectors, which may operate concurrently with mutators, we need to avoid interference from delayed processes. We use a shared variable `shRnd` to hold the round number of the current GC, together with an additional field `round` in the record of a node. The private variable *rnd* is a private copy of the shared variable `shRnd`. A collector p participates in the current round of GC if and only if $rnd_p =$ `shRnd`. We introduce the global private variable *toBeC* to transfer information about checked nodes between internal calls. There is also a local private variable *toBeD* in procedure *GCollect*.

3.2. Algorithm

In this section, we give the high-level implementation for the collectors and the mutators. All atomic commands (regions) are labeled with a number. It is well known that, since private activity cannot lead to interference with other processes, actions on private variables can be freely merged to one of the nearest atomic regions without violating the atomicity restriction, e.g., see [1] Theorem 6.26. We use the atomicity brackets ⟨ ⟩ only when the region refers to a shared variable more than once.

Since procedure calls only modify private control data, procedure headers are not always numbered themselves, but their bodies usually have numbered atomic statements. The location numbers are chosen identical to the numbers in the PVS code, and are therefore not completely consecutive.

Brackets ⟦ ⟧ and the actions between braces { } and parentheses (| |) can be ignored in the implementation. They only serve in the proof of correctness. We will explain this in Section 4.

```
proc GCollect() =
    local x : Index; toBeD : subset of Index;
% first phase
100: rnd := shRnd; toBeC := Index;
101: while shRnd = rnd ∧ toBeC ≠ ∅ do
        choose x ∈ toBeC;
108:    ⟨ if round[x] = rnd then
            round[x] := rnd + 1; ari[x] := arity[x]; { inGC[x] := true; }
            if color[x] = black then color[x] := grey; fi;
            if srcnt[x] > 0 then father[x] := 0; else father[x] := −1; fi; fi; ⟩
        toBeC := toBeC \ {x};
    od;
% second phase
121: toBeC := Index; toBeD := Index;
122: while shRnd = rnd ∧ toBeD ≠ ∅ do
        choose x ∈ toBeD;
126:    toBeD := toBeD \ {x};
        ⟨ if father[x] = 0 then ⟩
            Mark_stack(x); fi;
    od;
% third phase
129: while shRnd = rnd ∧ toBeC ≠ ∅ do
        choose x ∈ toBeC;
134:    ⟨ if round[x] = rnd + 1 ∧ color[x] = grey then
              color[x] := white;
              (| assert ¬R(x) ∧ x ∉ free; free := free ∪ x; |) fi; ⟩
        toBeC := toBeC \ {x};
    od;
135: ⟨ if rnd = shRnd then shRnd := rnd + 1; { inGC := λ(i ∈ Index) : false; } fi; ⟩
137: return
end GCollect.
```

Fig. 4. Procedure *GCollect*.

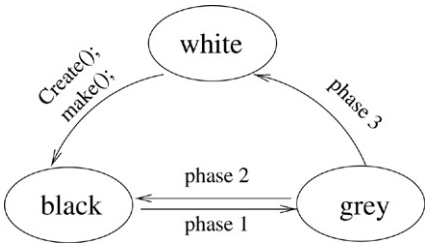

Fig. 5. Transitions between colors.

3.2.1. Collectors

Our garbage collectors are encoded in the procedure *GCollect* as shown in Fig. 4. This procedure calls procedure *Mark_stack* shown in Fig. 6. As announced above, GC consists of three phases: (1) paint all *black* nodes *grey* while recording the current memory structure, (2) paint all *grey* nodes reachable from the source nodes back to *black* after traversing the memory graph, and (3) reclaim all garbage by painting all remaining *grey* nodes *white*. The transitions between the colors are shown in Fig. 5.

Collectors first let *rnd* get the current value of `shRnd` (this is the only action that updates the private variable *rnd*) to prepare for participating in this round of GC. A new round of GC is started when the fastest collector reaches location 101 with $rnd_{self} = \texttt{shRnd}$ holding in the precondition. It is proved by means of invariants that before a new round of GC is started, all earlier rounds of GC have completed:

$$\forall x \in \texttt{Index} : \texttt{round}[x] = \texttt{shRnd} \wedge \texttt{color}[x] \neq grey.$$

```
proc  Mark_stack (x : Index) =
      local w, y : Index; suc : Bool; j, k : ℕ;
      stack : Stack; head : ℕ; set : subset of Index;
      ch : array [1 ... C] of Index;
150:  toBeC := toBeC \ {x}; set := {x}; head := 0;
151:  while shRnd = rnd ∧ set ≠ ∅ do
        choose w ∈ set;
157:    set := set \ {w};
        ⟨ if color[w] = grey ∧ round[w] = rnd + 1 then
            k := ari[w];
            for j := 1 to k do ch[j] := child[w, j] od; ⟩
            head++; stack[head] := w; j := 1;
158:        while shRnd = rnd ∧ j ≤ k do
              y := ch[j];
              if y ∉ toBeC then j++;
              else
163:            ⟨ if (father[y] = −1 ∨ father[y] = w)
                        ∧ color[y] = grey ∧ round[y] = rnd + 1 then
                    father[y] := w; ⟩
                    toBeC := toBeC \ {y}; set := set ∪ {y}; fi;
                  j++; fi;
            od; fi;
      od;
168:  while shRnd = rnd ∧ head ≠ 0 do
        y := stack[head];
175:    head--;
        ⟨ if color[y] = grey ∧ round[y] = rnd + 1 then
            color[y] := black;
            srcnt[y] := srcnt[y] − freecnt[y]; freecnt[y] := 0; fi; ⟩
      od;
180:  return
end Mark_stack.
```

Fig. 6. Procedure *Mark_stack*.

In order to prevent some collector from doing useless or even harmful work, every modification on a node in each phase is protected by a guard, which forces the collectors with $rnd_{self} \neq$ shRnd to abandon their delayed activity.

In the first phase, from location 101 to location 108, collectors try to update field round, paint *black* nodes *grey* and record the present memory structure using fields ari and father. The collectors only need to paint the *black* nodes *grey* since the *white* nodes can not be garbage.

As the algorithm allows parallel use of mutators, being a source node is not stable during GC. A new source node can be allocated from the free set by *Create* or *Make*, or generated by *Protect* or *Send* during GC.

There may be some delay in decrementing field srcnt when the number of references decreases (see *UnProtect*). Therefore, we can not say a node x is a source node if its field srcnt is positive. The fact is that srcnt of a node is positive if it has ever been a source node in the period since the latest execution of location 175.

We let the field father of each node with positive srcnt be 0, and that of other nodes be -1 in the first phase. A new source node x can then be distinguished from others by checking if $\texttt{srcnt}[x] > 0 \wedge \texttt{father}[x] \neq 0$ holds. For simplicity, we say that a node x with $\texttt{father}[x] = 0$ is an *old source node*. When the fastest collector participating in the current GC is at the end of its first phase, all nonfree nodes are *grey* except that new source nodes are *black*.

A delayed initialization on node x will be skipped because of the guard in location 108 since round$[x]$ is never decreased. As usual with version numbers, here we need to assume that sufficient bits are allocated for the version numbers to ensure that they cannot "wrap around" during the interval of a process's GC cycle.

In the second phase, from location 121 to location 126, the collectors build a forest in the set of all reachable nodes starting from the old source nodes. Trees in the forest are mutually disjoint. Each of them is rooted by a chosen old

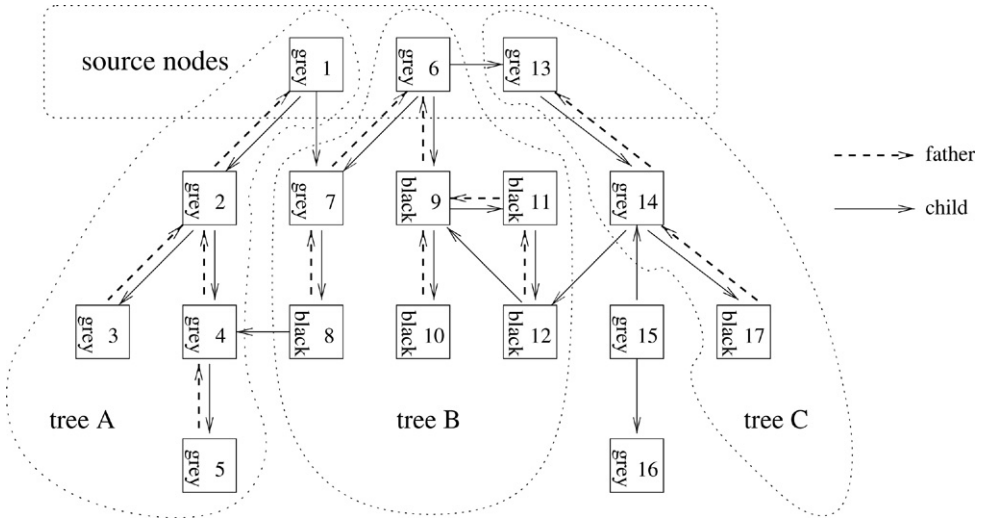

Fig. 7. A garbage collection scenario.

source node, and is created via calling a procedure *Mark_stack* (see Fig. 6) in a *while* loop. During *Mark_stack*, all the *grey* nodes on the tree are painted back to *black* in the order from the leaves to the root.

The procedure *Mark_stack* is mainly a form of graph search, and it was initially designed as a recursive procedure. Since we wanted to prove the correctness of our algorithm with PVS, we eliminated the recursion in favor of an explicit stack. The private variable *toBeC* serves to ensure that the search of a collector traverses every node at most once. This is important since the memory graph may have cycles and nodes may be reachable from different old source nodes.

In *Mark_stack*, from location 151 to location 163, the tree (in the forest) is formed by setting the `father` pointers. Since the memory graph is not a tree and may even have cycles, the collectors must reach consensus about the tree. The collectors starting from the same old source node cooperate with each other, and are in competition with others to expand the tree to all nodes reached. E.g. in the scenario of Fig. 7, node 7 belongs to tree *B* since one of the collectors forming tree *B* first detects that node and sets its `father` to 6. The collectors forming tree *A* will ignore node 7 since its `father` is now neither -1 nor in tree *A*. Note that all slower processes starting from the same old source node use the same tree for tracing reachable nodes if the task is not finished. The tree stops growing when every leaf node has no child that regards it as its father.

The order for choosing an element from the local variable *set* is irrelevant for correctness, but relevant for efficiency. The search is a depth first search if the order is first in last out. The search is a breadth first search if the order is first in first out.

The reduction theorems require that every atomic region of the high-level algorithm refers to at most one shared node. In the procedure *Mark_stack*, local variables `ch` and *k* are therefore introduced to temporarily store the old children of a node. This also prevents collectors from visiting a shared node unnecessarily. It adds a proof obligation that these local variables preserve the information of the node when the process is not delayed.

Starting from the chosen old source node, all nodes on the tree are pushed on the local stack after their old children have been temporarily stored. The order of the elements pushed on the stack is essential for correctness.

After the tree has been established, the collector paints all *grey* nodes *black* in the order in which they are popped from the *stack* (from location 168 to location 175) if the action is not too late. When a node in the tree is painted *black*, its descendants (with respect to the *father* relation) in the tree have been painted *black* already (see Fig. 7). So the other collectors need not trace or paint the subtree starting from that node. In particular, collectors need not trace or paint the tree starting from a new source node. The proof of all this requires interesting and rather complicated graph theoretic invariants. At the end of *Mark_stack*, the process returns to the procedure *GCollect* to search the tree from another old source node.

Note that it is sufficient to explore all accessible *grey* nodes in the second phase without the help of new source nodes. Using the view of the memory structure taken in the first phase may cause it to miss collecting some new

garbage that is generated by *UnProtect*ing a source node after the first phase, but this does not matter since the new garbage will be recycled within two rounds of GC according to the liveness property (we will come to this later).

All old source nodes appear in the different trees of the forest. The tasks of tracing reachable nodes starting from the different old source nodes can be distributed among several processes. When the fastest collector is at the end of the second phase, all accessible *grey* nodes have been detected and painted *black*.

In the third phase, from location 129 to the end of the procedure, collectors try to recycle all remaining *grey* nodes by coloring them *white* (i.e., adding them to the `free` set). The main proof obligation for the algorithm is that all nodes being freed are not accessible. When the fastest collector is at the end of the third phase, it increments the shared variable `shRnd` in location 135 to notify all other collectors to quit garbage collecting. At that point, there are no grey nodes, see invariant *I66* in Appendix A. We define a round of GC to be completed when this fastest process executes location 135.

It should be noticed that all GC modifications of the graph are in the atomic regions numbered 108, 134, 157, 163, 175, which are guarded by conditions containing something like $\texttt{round}[x] = rnd$ or $\texttt{round}[x] = rnd + 1$. These conditions preclude delayed collectors from destructive interferences, but we needed the PVS verification to convince ourselves that they are indeed sufficient for this purpose. On the other hand, the tests $\texttt{shRnd} = rnd$ in 101, 122, 129, 151, 158, 168 are superfluous for correctness and merely serve to terminate innocent but useless activity.

It is advantageous that collectors may exchange information. The collectors involved in the same round of GC should not use the same strategy for choosing x in the same phase. For the interested reader, more details can be found in the algorithms for the *write-all-problem* [23,37]. The main idea is to partition the task statically into many roughly equivalent subtasks (more subtasks than the number of available threads), and then let each thread dynamically claim one subtask at a time and remove the subtask after completion.

3.2.2. Mutators

The implementations of the procedures for the mutators are relatively easy. We provide the code in Figs. 8 and 9 for the interface procedures in the mutators, which match directly with the procedures in the specification. Note that the mutators do not modify fields `ari`, `father` and `round` of nodes.

Procedures *Create* and *Make* serve to extend the memory graph with a new node. In *Create* and *Make*, "time to do GC" indicates that some variable, like time or the amount of free memory, reaches a threshold value. Allocation in the mutator (see *Create* and *Make*) is potentially expensive. It requires a linear search over the whole memory. This problem can be solved by implementing the free set as a lock-free list (see [44,55]) with adding a new element to the list in a new numbered location just before the last **fi** in location 134, and deletions of elements from the list in locations 200 and 300.

In procedure *UnProtect*, at location 450, the decrementing of the field `srcnt` of the node is postponed when the mutator removes the node from its *roots* set. Instead, we use the field `freecnt` to count every delayed *UnProtect*. The immediate incrementation of `srcnt` is incorrect because of the following counterexample. Assume there are three nodes: node 1 is a free node, node 2 is a source node, with one child, node 3. Now, collector p starts the first phase of GC. Just after collector p executes 108 at node 1, which is white, it goes to sleep. Then mutator q is scheduled and *Make*s node 1 a new root node, of which the `color` becomes *black* (instead of *grey*), and sets node 3 as a new child of node 1. Then mutator q *UnProtect*s node 2, and node 2 happens to become a non-source node afterwards. Then collector p wakes up, resumes executing 108 at node 2 and node 3. Since in the second phase collectors only explore all *grey* nodes reachable from old source nodes, they will regard node 3 as an inaccessible node and collect it mistakenly as garbage in the third phase.

One may wonder why the decrementing of the field `srcnt` is postponed from *UnProtect* to location 175 of *Mark_stack*. We tried to update fields `srcnt` in the first phase of GC. However, we found that this is not correct while we proceeded the mechanical proof with PVS. The counterexample is the same as the previous one. After inspecting some invariants, we found that all accessible *grey* nodes can be traced without the help of either the *black* nodes or the upper *grey* nodes resided in the local stack. This means that it is safe to update the field `srcnt` at that moment. Moreover, fields `srcnt` of all remaining *grey* nodes appearing in the third phase are all zero and therefore need not be decremented.

```
proc  Create() : Index =
      local x : Index;
      while true do
200:     choose x ∈ Index;
206:     ⟨ if color[x] = white then
             color[x] := black; srcnt[x] := 1;
             (| assert x ∈ free; free := free \ x; |)
             [[ arity[x] := 0; roots := roots ∪ {x}; ]] ⟩
             break;
208:     elseif time to do GC then
             GCollect(); fi;
      od;
210:  [[ return x ]]
end Create.

proc  AddChild(x, y : Index) : Bool =
{ R(self, x) ∧ R(self, y) }
      local suc : Bool;
258:  ⟨ [[ suc := (arity[x] < C);
         if suc then arity[x]++; child[x, arity[x]] := y; fi ]] ⟩
262:  [[ return suc ]]
end AddChild.

proc  GetChild(x : Index, rth : 1...C) : Index ∪ {0} =
{ R(self, x) }
      local y : Index ∪ {0};
280:  ⟨ [[ if 1 ≤ rth ≤ arity[x] then y := child[x, rth]; else y := 0; fi ]] ⟩
284:  [[ return y ]]
end GetChild.

proc  Make(c : array [ ] of Index, n : 1...C) : Index =
{ ∀ j ∈ [1...n] : R(self, c[j]) }
      local x : Index; j : ℕ;
      while true do
300:     choose x ∈ Index;
306:     ⟨ if color[x] = white then
             color[x] := black; srcnt[x] := 1;
             (| assert x ∈ free; free := free \ x; |)
             [[ for j := 1 to n do child[x, j] := c[j]; od
               arity[x] := n; roots := roots ∪ {x}; ]] ⟩
             break;
308:     elseif time to do GC then
             GCollect(); fi;
      od;
310:  [[ return x ]]
end Make.
```

Fig. 8. Procedures *Create*, *AddChild*, *GetChild*, and *Make*.

In procedure *Send* and *Receive*, the weaker requirement on the reference counter (i.e., field `srcnt` of a node) is based on the fact that the reference counter does not always need to be accurate.

4. Correctness

The main issue of the algorithm is how to ensure the correct execution of collectors and mutators when they concurrently compete with each other for the same data structure. The standard notion of correctness for asynchronous parallel algorithms is to assume that the atomic instructions of the threads are interleaved in an arbitrary linear order. The algorithm is correct if it behaves properly for all such interleavings. Any property can be considered as the

```
proc Protect(x : Index) =
{ R(self, x) ∧ x ∉ roots }
400:  ⟨ srcnt[x]++; ⟩
      ⟦ roots := roots ∪ {x}; ⟧
408:  ⟦ return ⟧
end Protect.

proc UnProtect(z : Index) =
{ z ∈ roots }
450:  ⟨ freecnt[z]++; ⟩
      ⟦ roots := roots \ {z}; ⟧
460:  ⟦ return ⟧
end UnProtect.

proc Send(x : Index, r : Process) =
{ R(self, x) ∧ Mbox[self, r] = 0 }
500:  ⟨ srcnt[x]++; ⟩
508:  ⟦ Mbox[self, r] := x; ⟧
510:  ⟦ return ⟧
end Send.

proc Receive(r : Process) : Index =
{ Mbox[r, self] ≠ 0 }
      local x : Index;
550:  ⟦ x := Mbox[r, self]; ⟧
552:  if x ∉ roots then
        ⟦ Mbox[r, self] := 0; roots := roots ∪ {x}; ⟧
      else
554:    ⟨ srcnt[x]--; ⟩
559:    ⟦ Mbox[r, self] := 0; ⟧ (| assert x ∈ roots; |) fi;
560:  ⟦ return ⟧
end Receive.

proc Check(r, q : Process) : Bool
      local suc : Bool;
600:  ⟦ suc := (Mbox[r, q] = 0); ⟧
602:  ⟦ return suc ⟧
end Check.
```

Fig. 9. Procedures *Protect*, *UnProtect*, *Send*, *Receive*, and *Check*.

conjunction of safety properties and liveness properties. In this section we describe the proofs of safety properties and a liveness property of the algorithm by means of invariants.

4.1. Modeling the mutators

In order to verify our memory management system in PVS, we model the mutators very nondeterministically in the following loop that may call the interface procedures in arbitrary order and with arbitrary arguments provided the preconditions are met. This is not part of the memory management system itself, and therefore not to be implemented. It is used in the PVS proof to verify the correctness of the system under all possible applications, in the same way as, e.g., in [29] Section 4.2. Here we use the operator [] to indicate a nondeterministic choice. It binds weaker than the semicolon of sequential composition.

```
      loop
1:        Create()
        [] choose x, y with R(self, x) ∧ R(self, y) ; AddChild(x, y)
        [] choose x, rth with R(self, x) ; GetChild(x, rth)
        [] choose c, n with (∀ j ∈ [1 . . . n] : R(self, c[j])) ; Make(c, n)
        [] choose x with R(self, x) ∧ x ∉ roots ; Protect(x)
```

```
  []  choose x ∈ roots ; UnProtect(x);
  []  choose x, r with R(self, x) ∧ Mbox(self, r) = 0 ; Send(x, r)
  []  choose r with Mbox(r, self) ≠ 0 ; Receive(r)
  []  choose q, r ; Check(q, r)
endloop
```

Normally, after some operation is finished, the mutator will return to the main loop. In the implementation, there are two places where a mutator is temporarily allowed to become a collector by calling *GCollect*. We introduce an auxiliary private variable *return* to hold the return location. Since they are private, they can be assumed to be touched instantaneously without violation of the atomicity restriction.

4.2. Safety properties

The main aspect of safety is functional correctness and atomicity, say in the sense of [41]. We prove correctness of the implementation by showing that each procedure of the implementation executes its specification command always exactly once and that the resulting value of the implementation equals the resulting value in the specification. As shown in Figs. 4–9, we therefore extend the implementations with auxiliary variables and commands used in the specification. We use brackets ⟦ ⟧ to enclose implementation commands that perform the same actions as the specification. Specification commands that are deleted in implementation are enclosed between ⟪ ⟫.

GC should be an internal affair that is functionally equivalent to *skip*. The main safety property of *GCollect* is that it only collects garbage, i.e., that an accessible node is never freed. This is expressed in the invariant:

I1: $white(x) \Rightarrow \neg R(x)$.

Here and henceforth, we write $white/(x)$ for $\texttt{color}[x] = white$, and similarly for the other two colors.

The implementation is an extension of the specification except that the specification variable `free` is the set of the *white* nodes of the implementation. Apart from the common actions enclosed in ⟦ ⟧, all implementation commands do not modify the specification variables and all specification commands do not modify the implementation variables. We therefore do not distinguish the variables and commands common to both specification and implementation, and enclose them between ⟦ ⟧.

Functional correctness of the mutator procedures now follows from the invariants:

I2: $white(x) \equiv x \in \texttt{free}$
I3: $554 \leq pc_p \leq 559 \Rightarrow x_p \in roots_p$.

Indeed, by removing the implementation variables from the combined program, we obtain the specification. This removal eliminates many atomic steps of the implementation. This is known as removal of stutterings in TLA [38] or abstraction from τ steps in process algebras. In the cases of *Create* and *Make*, the guard is translated by means of the invariant *I2*. In the case of *Receive*, we use invariant *I3* to justify the change in the control flow.

In order to prove that *I1*, *I2*, and *I3* are indeed invariants, we had to invent the invariants listed in Appendix A. More specifically, Appendix B shows that *I1* follows from the invariants *I3*, *I5*, *I18*, *I71*, which can all be found in Appendix A. Indeed, *I18* is *I1* with relation *R* replaced by *R1*, and the other three invariants are used to prove that $R(x)$ implies $R1(x)$. As for *I18* itself, Appendix B shows that it is preserved in every step of the algorithm provided the predicates *I6*, *I8*, *I9*, *I12*, *I16*, *I25*, *I64*, and *I69* hold in the precondition. To show that *I3* is an invariant, we use *I28* in the precondition, and so on.

Fortunately, this process of inventing invariants terminates. For this to happen, however, we needed to introduce an auxiliary shared variable *inGC* to indicate which nodes are involved in the current round of GC. All operations on *inGC* are enclosed in braces { }, and can be assumed to be executed instantaneously without violation of the atomicity restriction. For the role of *inGC*, see the invariants in Appendix A, in particular, e.g., *I14*, *I17*, etc. The use of auxiliary variables goes back to [49]. For validity we refer to [1] Lemma 7.3.

An important class of invariants are those that assert that the preconditions of the mutator procedures are stable under the actions of the other processes. For *AddChild*, *GetChild*, *Make*, *Protect*, *Send* and *Receive*, respectively, these stability conditions are expressed by the invariants:

I6: $250 \le pc_p \le 258 \;\Rightarrow\; R(p,\ x_p) \wedge R(p,\ y_p)$
I7: $pc_p = 280 \;\Rightarrow\; R(p,\ x_p)$
I8: $300 \le pc_p \le 308 \vee (100 \le pc_p \le 180 \wedge return_p = 300)$
$\Rightarrow\; (\forall k \in [1 \ldots n_p] : R(p,\ c_p[k]))$
I9: $pc_p = 400 \vee 500 \le pc_p \le 508 \;\Rightarrow\; R(p,\ x_p)$
I10: $500 \le pc_p \le 508 \;\Rightarrow\; \texttt{Mbox}[p,\ r_p] = 0$
I11: $550 \le pc_p \le 559 \;\Rightarrow\; \texttt{Mbox}[r_p,\ p] \neq 0.$

Any mutator, p, can ensure its rights of access to some node x by verifying $R(p, x)$ independently, because of the following lemma that asserts that $R(p, x)$ can only be invalidated by process p itself:

V1: $R(p,\ x) \wedge I18 \wedge I25 \wedge p \neq q \quad \triangleright_q \quad R(p,\ x),$

where we write $P \ \triangleright_q \ Q$ to express that, if precondition P holds and process q performs an atomic action, this action has postcondition Q.

As we announced earlier, no node is *grey* when the current round of GC is finished. This is formalized in the following invariant:

I4: $grey(x) \;\Rightarrow\; (\exists\ p : rmd_p = \texttt{shRnd})$

where $rmd_p = \texttt{shRnd}$ indicates that process p is involved in the current round of GC.

For any node x, the difference $\texttt{srcnt}[x] - \texttt{freecnt}[x]$ counts the number of references to x as a source node. Since an atomic region in the high-level implementation must not refer to different shared variables (this is an important requirement for the final lock-free transformation), values of a node and a mailbox can not be simultaneously modified in the same atomic region. The counter is precisely described by the following invariant:

I5: $\texttt{srcnt}[x] - \texttt{freecnt}[x] = \sharp(\{p \mid x \in roots_p\}) + \sharp(\{(p,\ q) \mid$
$(\texttt{Mbox}[p,\ q] = x \wedge \neg(pc_q = 559 \wedge p = r_q)) \vee (pc_p = 508 \wedge x_p = x \wedge q = r_p)\}).$

All the safety properties (invariants) have been proved with the interactive proof checker PVS. The use of PVS did not only take care of the delicate bookkeeping involved in the proof, it could also deal with many trivial cases automatically. At several occasions where PVS refused to complete the proof, we actually found some mistakes and had to correct previous versions of this algorithm. To prove these invariants, we need many other invariants. All proved invariants and lemmas are listed in Appendix A. Appendix B gives the dependencies between the invariants. For the complete mechanical proof, we refer the interested reader to [28].

4.3. Liveness

A liveness property asserts that program execution eventually reaches some desirable state. In our case, we want to ensure that every garbage node is eventually collected. We shall express this by means of the "*leads-to*" (denoted as $\rightsquigarrow$) relation that was developed for UNITY in [9]. For predicates P and Q, the assertion $P \rightsquigarrow Q$ is defined to mean $\Box(P \Rightarrow \Diamond Q)$, it is always the case that P implies eventually Q.

The liveness property of the algorithm we need to verify is that, always, every garbage node x is eventually collected:

$$\neg R(x) \rightsquigarrow white(x).$$

4.3.1. Auxiliary results about leads-to and unless

In order to prove the liveness property of the algorithm, we establish the needed techniques. First, we introduce fairness into our formalism. This can be done with a single rule: if some process is at the location of some atomic action, the process will eventually execute the action and arrive at the next location. We also assume that GC is infinitely often called during execution. This is formalized in the fairness assumption $\Box(\Diamond(\exists p : pc_p = 100))$.

Except some well-known lemmas extracted from the literature, all lemmas in this section have been verified mechanically with PVS. The following results are stated in [50].

Lemma 4.1. *For predicates P, Q, and R, we have*
(a) *Relation $\rightsquigarrow$ is reflexive and transitive.*
(b) *If $P \Rightarrow Q$ then $P \rightsquigarrow Q$.*
(c) *If $P \rightsquigarrow R$ and $Q \rightsquigarrow R$ then $(P \vee Q) \rightsquigarrow R$.*
(d) $P \rightsquigarrow (Q \vee (P \wedge \Box\neg Q))$.

Lemma 4.1(a), (b) and (c) are used to prove a general *proof lattice* for a program, which is addressed in [50]. Intuitively, Lemma 4.1(d) holds because starting in a state where P is true, either Q will be true in some subsequent state, or $\neg Q$ will be always true from then on. Thus, the general pattern of these proofs by contradiction is to assume that the desired predicate never becomes true, and then show that this assumption leads to a contradiction. For more details, refer to [50].

The "steps-to" relation $P \triangleright Q$ is defined to mean that, if P holds in the precondition of any atomic action, Q holds in the postcondition. The "unless" relation $\mathcal{U}$ is defined by

$$(P \,\mathcal{U}\, Q) \equiv P \triangleright (P \vee Q).$$

These relations are quite useful to prove "leads-to" ($\rightsquigarrow$) relations. Since they only involve a single step, they can be checked directly by PVS with the help of invariants. It is not hard to prove the following general lemmas, which are used in the proof of the liveness property.

Lemma 4.2. *For predicates P and Q, we have*
(a) *If P and $(P \rightsquigarrow Q)$ then $\Diamond Q$.*
(b) *If $P \,\mathcal{U}\, Q$ and $\Diamond\neg P$ then $P \rightsquigarrow Q$.*

Lemma 4.3. *Let $Q(w)$ be predicates for all $w \in I$, where I is a finite set. Let P, R, S and T be predicates satisfying the following three assumptions:*
(1) $P \rightsquigarrow T \vee (S \wedge R \wedge Q(w))$ *for all $w \in I$,*
(2) $(S \wedge R \wedge Q(w)) \;\mathcal{U}\; (T \vee \neg S)$ *for all $w \in I$,*
(3) $\neg S \triangleright \neg S$.
Then $P \rightsquigarrow T \vee (S \wedge R \wedge \forall\, w \in I : Q(w))$.

The next lemma expresses that we may always introduce or delete the invariants of the system.

Lemma 4.4. *Let J be an invariant, and P and Q be predicates. Then $P \wedge J \rightsquigarrow Q$ implies $P \rightsquigarrow Q \wedge J$, and $(P \wedge J) \,\mathcal{U}\, Q$ implies $P \,\mathcal{U}\, (Q \wedge J)$.*

4.3.2. Every garbage node is collected within two rounds

We prove something stronger than suggested above, viz., that every inaccessible node is painted *white* within two rounds of GC. One round will take care of the case $\neg R1(x)$, whereas a second round is needed to reduce $\neg R(x)$ to $\neg R1(x)$.

Theorem 4.1. *For any integer m and node x,*

$$\neg R(x) \wedge \texttt{shRnd} = m \;\rightsquigarrow\; \mathit{white}(x) \wedge \texttt{shRnd} \leq m + 2.$$

The proof of this theorem is postponed to the end of this subsection. We first give some auxliary observations, definitions, and lemmas.

We need only consider states that are reachable from initial states, where therefore all invariants hold. By Lemma 4.4, we are therefore allowed to add to any predicate any conjunction of invariants at any time. According to *I12* and *I68*, every process p always has $md_p \leq \texttt{shRnd}$, while equality implies that $101 \leq pc_p \leq 180$. We now define predicates that express whether the fastest garbage collecting process is idle, or has arrived in the first phase, or the second phase, or the third phase, or at location 135, respectively, by:

$$\begin{aligned} A_0 &\equiv (\forall p : md_p < \texttt{shRnd}),\\ A_1 &\equiv (\exists p : pc/_p \in [101, 110] \wedge md_p = \texttt{shRnd})\\ &\qquad \wedge \neg(\exists p : pc/_p \notin [101, 110] \wedge md_p = \texttt{shRnd}), \end{aligned}$$

$$
\begin{aligned}
A_2 &\equiv (\exists p : pc/_p \notin [101, 110] \wedge md_p = \mathtt{shRnd}) \\
&\quad \wedge \neg(\exists p : pc/_p \in [129, 135] \wedge md_p = \mathtt{shRnd}), \\
A_3 &\equiv (\exists p : pc/_p \in [129, 135] \wedge md_p = \mathtt{shRnd}) \\
&\quad \wedge \neg(\exists p : pc/_p = 135 \wedge md_p = \mathtt{shRnd}), \\
A_4 &\equiv (\exists p : pc/_p = 135 \wedge md_p = \mathtt{shRnd}).
\end{aligned}
$$

Since we also need to keep track of the value of shRnd, we define $A_i(m) \equiv (A_i \wedge \mathtt{shRnd} = m)$. These predicates are mutually exclusive and partition the state space. They satisfy the following unless relations:

$$A_0(m)\ \mathcal{U}\ A_1(m)\ \mathcal{U}\ A_2(m)\ \mathcal{U}\ A_3(m)\ \mathcal{U}\ A_4(m)\ \mathcal{U}\ A_0(m+1).$$

To prove Theorem 4.1, we first investigate what may happen to a black node x that is not accessible according to *R1* when $A_0(m)$ holds. Notice that this implies $\mathtt{round}[x] = m$ by the invariants *I13* and *I15*. Part (a) of Lemma 4.5 implies that the next GC round paints such a node x white. The other parts are needed for the other phases $A_i(m)$ and the cases with $\neg R(x)$.

Lemma 4.5. *For any integer m and nodes x and w,*

(a) $A_0(m) \wedge \neg R1(x) \wedge black(x)$
$\mathcal{U}\ A_1(m) \wedge \neg R1(x) \wedge black(x) \wedge \mathtt{round}[x] = m$
$\mathcal{U}\ A_1(m) \wedge \neg R1(x) \wedge grey(x) \wedge \mathtt{round}[x] = m + 1$
$\mathcal{U}\ A_2(m) \wedge \neg R1(x) \wedge grey(x) \wedge \mathtt{round}[x] = m + 1$
$\mathcal{U}\ A_3(m) \wedge \neg R1(x) \wedge grey(x) \wedge \mathtt{round}[x] = m + 1$
$\mathcal{U}\ A_3(m) \wedge white(x)$

(b) $\mathtt{shRnd} = m = \mathtt{round}[x] - 1 \wedge \neg R1(x) \wedge \neg white(x)$
$\mathcal{U}\ (\mathtt{shRnd} = m \wedge white(x)) \vee (\mathtt{shRnd} = m + 1 = \mathtt{round}[x] \wedge \neg R1(x) \wedge \neg white(x))$

(c) $A_0(m) \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge black(w) \wedge \mathtt{srcnt}(w) > 0$
$\mathcal{U}\ A_1(m) \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge black(w) \wedge \mathtt{srcnt}(w) > 0 \wedge \mathtt{round}[w] = m$
$\mathcal{U}\ A_1(m) \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge grey(w) \wedge \mathtt{srcnt}(w) > 0 \wedge \mathtt{round}[w] = m + 1$
$\mathcal{U}\ A_2(m) \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge grey(w) \wedge \mathtt{srcnt}(w) > 0 \wedge \mathtt{round}[w] = m + 1$
$\mathcal{U}\ A_2(m) \wedge \neg R(x) \wedge \neg white(x) \wedge \mathtt{srcnt}(w) = 0$

(d) $\mathtt{shRnd} = m = \mathtt{round}[w] - 1 \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge \mathtt{srcnt}(w) > 0$
$\mathcal{U}\ (\mathtt{shRnd} = m \wedge white(x)) \vee (\mathtt{shRnd} = m \wedge \neg R(x) \wedge \neg white(x) \wedge \mathtt{srcnt}(w) = 0)$
$\vee (\mathtt{shRnd} = m + 1 = \mathtt{round}[w] \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge \mathtt{srcnt}(w) > 0)$

(e) $\mathtt{shRnd} \leq m \wedge \neg R(x) \wedge \neg white(x) \wedge \neg(\mathtt{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))$
$\mathcal{U}\ (\mathtt{shRnd} \leq m \wedge white(x)) \vee \mathtt{shRnd} > m$

(f) $\mathtt{shRnd} > m \ \triangleright\ \mathtt{shRnd} > m.$

This lemma has been verified with PVS.

On the other hand, we have the following simple progress result.

Lemma 4.6. *For any integer m, we have* $\mathtt{shRnd} = m \ \leadsto\ \mathtt{shRnd} = m + 1$.

Proof. We first prove that $A_4(m) \ \leadsto\ \mathtt{shRnd} = m + 1$. In fact, the shared variable shRnd can be modified only by some process executing location 135 with precondition $md_{self} = \mathtt{shRnd}$. Since $A_4(m)$ implies that there is indeed such a process, it follows that shRnd will be eventually incremented by 1 according to fairness.

By transitivity of $\leadsto$, it now suffices to prove $\mathtt{shRnd} = m \ \leadsto\ A_4(m)$. In view of Lemma 4.1(d), we may assume $\Box\neg A_4(m)$. Since the shared variable shRnd can be modified only by some process executing location 135 with precondition $md_{self} = \mathtt{shRnd}$, we then obtain that shRnd remains constant, i.e., m. By assumption, there will be eventually some process p with $pc_p = 100$. Because of the fairness of atomic actions, we then get $\Diamond(md_p = \mathtt{shRnd} = m \wedge pc_p = 101)$. Since *toBeC* and *toBeD* are both private variables, all loops in GC are finite (see procedures *GCollect* and *Mark_stack*). We therefore obtain $\Diamond(md_p = \mathtt{shRnd} = m \wedge pc_p = 135)$ according to fairness. This leads to a contradiction. □

Corollary 4.1. *In Lemma 4.5, all "unless" ($\mathcal{U}$) relations can be replaced by "leads-to" ($\leadsto$) relations.*

Proof. By Lemmas 4.2(a) and 4.6, we obtain $\Diamond(\texttt{shRnd} \neq m)$. Therefore, this corollary follows from Lemma 4.2(b). □

Corollary 4.2. *For any integer m and node x,*

$$shRnd = m = round[x] \wedge \neg R1(x) \wedge \neg white(x) \rightsquigarrow shRnd = m \wedge white(x).$$

Proof. By invariant *I16*, we obtain $black(x)$. By invariant *I34*, we have $A_0(m) \vee A1(m)$. Using transitivity of the "leads-to" relation, this follows from Lemma 4.5(a) and Corollary 4.1. □

Corollary 4.3. *For any integer m and node x,*

$$shRnd = m \wedge \neg R1(x) \wedge \neg white(x) \rightsquigarrow shRnd \leq m + 1 \wedge white(x).$$

Proof. By invariant *I13*, $\texttt{shRnd} = m$ implies $\texttt{round}[x] = m \vee \texttt{round}[x] = m + 1$. Therefore, the assertion follows from Lemma 4.5(b) and Corollaries 4.1 and 4.2. □

Corollary 4.4. *For any integer m and nodes x and w,*

$$\neg R(x) \wedge \neg white(x) \wedge shRnd = m = round[w] \wedge \neg R(w) \wedge srcnt(w) > 0 \rightsquigarrow shRnd = m \wedge \neg R(x) \wedge \neg white(x) \wedge srcnt(w) = 0.$$

Proof. Since $\texttt{srcnt}(w) > 0$, we have $R1(w)$. By invariants *I16* and *I18*, we then obtain $black(w)$. By invariant *I34*, we have $A_0(m) \vee A_1(m)$. Now using transitivity of the "leads-to" relation, the assertion follows from Lemma 4.5(c) and Corollary 4.1. □

Corollary 4.5. *For any integer m and nodes x and w,*

$$shRnd = m \wedge \neg R(x) \wedge \neg white(x) \rightsquigarrow (shRnd = m \wedge white(x)) \vee (shRnd \leq m + 1 \wedge \neg R(x) \wedge \neg white(x) \wedge \neg(srcnt(w) > 0 \wedge (w \xrightarrow{*} x)))$$

Proof. In view of the consequent, we clearly may assume that $\texttt{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x)$ holds initially. Then, by definition of relation R and transitivity of "$\xrightarrow{*}$", we obtain $\neg R(w)$. By invariant *I13*, we have $\texttt{round}[w] = m \vee \texttt{round}[w] = m + 1$. Then, the assertion follows from Lemma 4.5(d) and Corollaries 4.1 and 4.4. □

Corollary 4.6. *For any integer m and node x,*

$$shRnd = m \wedge \neg R(x) \wedge \neg white(x) \rightsquigarrow shRnd \leq m + 1 \wedge (white(x) \vee \neg R1(x)).$$

Proof. We use Lemma 4.3 with `Index` for I and the substitutions:

$$\begin{aligned} P &:= \texttt{shRnd} = m \wedge \neg R(x) \wedge \neg white(x), \\ T &:= \texttt{shRnd} \leq m + 1 \wedge white(x), \\ S &:= \texttt{shRnd} \leq m + 1, \\ R &:= \neg R(x) \wedge \neg white(x), \\ Q(w) &:= \neg(\texttt{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x)). \end{aligned}$$

The first premise of Lemma 4.3 holds because of Corollary 4.5. The second and third premises follow from Lemma 4.5(e) and (f). We finally simplify the consequent. □

Proof of Theorem 4.1. If $\neg R(x) \wedge \texttt{shRnd} = m$ holds and x is not already white, Corollary 4.6 implies that within one round, x is white or $\neg R1(x)$ holds. In the latter case, Corollary 4.3 implies that x becomes white within the next round. Lemma 4.1 takes care of the formal details. □

5. The low-level implementation

Synchronization primitives *load-linked*, LL and *store-conditional*, SC, proposed by Jensen et al. [34], have found widespread acceptance in modern processor architectures (e.g., MIPS II, PowerPC and Alpha architectures). These instructions are closely related to the CAS, and together implement an atomic Read/Write cycle. Instruction LL first reads a memory location, say X, and marks it as "reserved" (not "locked"). If no other processor changes the contents of X in between, the subsequent SC operation of the same processor succeeds and modifies the value stored; otherwise it fails. There is also a validate instruction VL, used to check whether X was not modified since the corresponding LL instruction was executed. Implementing VL is straightforward in an architecture that already supports SC. Note that the implementation does not access or manipulate X by other means than LL, SC or VL. Moir [45] showed that $LL/SC/VL$ can be constructed on any system that supports either LL/SC or CAS.

A shared variable X only accessed by $LL/SC/VL$ operations can be regarded as a variable that has an associated shared set of process identifiers $V.X$, which is initially empty. The semantics of LL, SC and VL are given by equivalent atomic statements below.

proc LL (**ref** X) : `value` =
  ⟨ $V.X := V.X \cup \{self\}$; **return** X; ⟩

proc SC (**ref** X; **in** Y) : `Bool` =
  ⟨ **if** $self \in V.X$ **then** $V.X := \emptyset$; $X := Y$; **return** *true*
    else return *false*; **fi** ⟩

proc VL (**ref** X) : `Bool` =
  ⟨ **return** $(self \in V.X)$ ⟩.

5.1. A pattern of general lock-free transformation

At the cost of copying an object's data before an operation, Herlihy [25] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives LL and SC. A process that needs access to a shared object pointed by X performs a loop of the following steps: (1) read X using an LL operation to gain access to the object's data area; (2) make a private copy of the indicated version of the object (this action need not be atomic); (3) perform the desired operation on the private copy to make a new version; (4) finally, call an SC operation on X to attempt to swing the pointer from the old version to the new version. The SC operation will fail when some other process has modified X since the LL operation, in which case the process has to repeat these steps until consistency is satisfied. The algorithm is non-blocking because at least one out of every P attempts must succeed within finite time. Of course, a process might always lose to some faster process, but this is unlikely in practice.

In [21], we formalize Herlihy's methodology [25] for transferring a sequential implementation of any data structure into a lock-free synchronization using synchronization primitives LL/SC, and develop a reduction theorem that enables us to reason about a general lock-free algorithm to be designed on a higher level than the synchronization primitives LL/SC.

The lock-free pattern we proposed in [21] is shown in Figs. 10 and 11, where the following statements are taken as a schematic representation of segments of code:

1. *noncrit* (**ref** *pub* : `aType`, *priv* : `bType`; **in** *tm* : `cType`; **out** $x : 1 \ldots N$): representing an atomic non-critical activity on variables *pub* and *priv* according to the value of *tm*, and choosing an index x of a shared node to be accessed.
2. *guard* (**in** X : `nodeType`, *priv* : `bType`) a non-atomic boolean test on the variable X of `nodeType`. It may depend on private variable *priv*.
3. *com* (**ref** X : `nodeType`; **in** *priv* : `bType`; **out** *tm* : `cType`): a non-atomic action on the variable X of `nodeType` and private variable *tm*. It is allowed to inspect private variable *priv*.
4. *read* (**ref** X : `nodeType`, **in** Y : `nodeType`): a non-atomic read operation that reads the value from the variable Y of `nodeType` to the variable X of `nodeType`, and does nothing else. If Y is modified during *read*, the resulting value of X is unspecified but type correct, and no error occurs.
5. LL, SC and VL: atomic actions as we defined before.

Shared variable
pub : aType;
Node : **array** [Index] **of** nodeType;
Private variable
priv : bType; *pc* : {*a1*, *a2*}; *x* : Index; *tm* : cType;

Program
loop
a1: *noncrit*(pub, *priv*, *tm*, *x*);
a2: ⟨ **if** *guard*(Node[*x*], *priv*) **then** *com*(Node[*x*], *priv*, *tm*); **fi**; ⟩
end.

Initial conditions
Θ_a : $\forall\, p \in \texttt{Process} : pc_p = a1$
Liveness
$\mathcal{L}_a$: $\Box(pc_p = a2 \longrightarrow \Diamond pc_p = a1)$

Fig. 10. Interface $\mathcal{S}_a$.

Constant P = total number of processes
Type
RefIndex $= 1 \ldots N + P$% recall that Index $= 1 \ldots N$
Shared variable
pub : aType;
node : **array** [RefIndex] **of** nodeType;
indir : **array** [Index] **of** RefIndex;
Private variable
priv : bType; *pc* : [*c1* ... *c7*];
x : Index; *mp*, *m* : RefIndex; *tm*, *tm1* : cType;

Program
loop
c1: *noncrit*(pub, *priv*, *tm*, *x*);
loop
c2: *m* := *LL*(indir[*x*]);
c3: *read*(node[*mp*], node[*m*]);
c4: **if** *guard*(node[*mp*], *priv*) **then**
c5: *com*(node[*mp*], *priv*, *tm1*);
c6: **if** *SC*(indir[*x*], *mp*) **then**
mp := *m*; *tm* := *tm1*; **break**; **fi**;
c7: **elseif** *VL*(indir[*x*]) **then break**; **fi**; **fi**;
end;
end.

Initial conditions
Θ_c : $(\forall\, p \in \texttt{Process} : pc_p = c1 \wedge mp_p = N{+}p) \wedge (\forall\, i \in \texttt{Index} : \texttt{indir}[i] = i)$
Liveness
$\mathcal{L}_c$: $\Box(pc_p = c2 \longrightarrow \Diamond pc_p = c1)$

Fig. 11. Lock-free implementation $\mathcal{S}_c$ of $\mathcal{S}_a$.

In the pattern, we are not interested in the internal details of these schematic commands but in their behavior with respect to lock-freedom.

As usual, the action enclosed by angular brackets $\langle \ldots \rangle$ is defined as atomic. The private variable x is intended only to determine the node under consideration, the private variable *tm* is intended to hold the result of the critical computation *com*, if executed.

We now need to fix the total number of processes, mutators and collectors together. We use P to stand for this number. In the concrete system $\mathcal{S}_c$ of Fig. 11, we declare P extra shared nodes for private use (one for each process).

Array `indir` acts as pointers to shared nodes. `node`[mp_p] can always be taken as a "private" node of process p though it is declared publicly: other processes can read it but cannot modify it. If some other process successfully updates a shared node while an active process p is copying the shared node to its "private" node, process p will restart the inner loop, since its private view of the node is not consistent anymore. After the assignment $mp := m$ at location $c6$, the "private" node becomes shared and the node shared previously (which contains the old version) becomes "private". Keep in mind that the composition of `node` and *indir* in $\mathcal{S}_c$ corresponds to `Node` in the abstract system $\mathcal{S}_a$ of Fig. 10.

The following theorem is the reduction theorem stated in [21].

Theorem 5.1. *The abstract system $\mathcal{S}_a$ defined in Fig. 10 is refined by the concrete system $\mathcal{S}_c$ defined in Fig. 11, i.e., there is a refinement mapping from $\mathcal{S}_c$ to $\mathcal{S}_a$.*

The reduction theorem is based on refinement mapping as described by Lamport [38], which asserts that a low-level specification correctly implements a high-level one. It has been verified with PVS. A reduction theorem is a general rule for deriving an "equivalent" high-level specification from a low-level one in some suitable sense [10].

5.2. The lock-free implementation

Refinement mappings enable us to reduce an implementation by reducing its components in relative isolation, and then gluing the *reductions* together with the same structure as the implementation. Atomicity guarantees that a parallel execution of a program gives the same results as a sequential and non-deterministic execution. This allows us to use the refinement calculus for stepwise refinement of transition systems [3]. Therefore, Theorem 5.1 can be universally employed for a lock-free construction to synchronize access to shared nodes of `nodeType`, and be sure that we end up with the reduction of the implementation. This allows us to design and verify a lock-free program on a higher level than the synchronization primitives. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level and thus the correctness can be more easily verified.

In the high-level implementation (from Fig. 4 to Fig. 9), instruction 135 is simply a CAS instruction offered by machine architectures or a Read/Write cycle that can easily be implemented by a LL/SC. Each of all other special commands enclosed by angular brackets $\langle \ldots \rangle$ only refer to one shared node and some private variables, and therefore can be transformed into low-level lock-free implementations using Theorem 5.1. E.g. location 108 of Fig. 4 is implemented in locations 104 ... 107 of Appendix C.2. Line 158 of Fig. 8 needs a simple program transformation before the pattern can be used. This results in locations 250 ... 256 of Appendix C.2.

At locations 126 and 157 (and possibly other cases), since these commands do not modify the node, swapping of pointers is unnecessary. We therefore use a simplified version where SC can be replaced by VL. The transformation is straightforward, and we present our final lock-free algorithm in Appendix C.

Apart from that, the high-level algorithm can also be transformed into a lock-free implementation by means of CAS using the other reduction theorem developed in [22]. This final transformation is a bit more complicated. Because of the similarity, we don't provide the theorem and the final transformation here.

6. Practical experiments

We carried out some experiments with our algorithm in order to show its feasibility and to obtain insight into its practical performance in the presence of several mutators and one designated collector. In case this collector does not have sufficient capacity, the mutators will automatically become collectors to offer a helping hand.

The main conclusion is that the performance of the algorithm is heavily influenced by its parameter settings such as the total number of nodes, the condition for joining the garbage collection process, the percentage of occupied nodes, the division of work between collectors and mutators, and the way storage of data leads to cache trashing. Understanding the trade-offs and finding optimal settings requires a study in itself, which is beyond the scope of this paper.

A second conclusion is that if we set these parameters well, we see no degrading in performance when increasing the number of mutators. A third conclusion is that if the number of processors is increased, performance decreases, most likely due to heavy interprocessor communication. A fourth conclusion is that due to the fact that garbage collection is a relatively elaborate affair, the performance in terms of the numbers of nodes that are created and collected per unit of time is relatively low.

Table 1
Some experimental results

P	1 processor				2 processors				4 processors			
	1		255		1		255		1		255	
	0	100	0	100	0	100	0	100	0	100	0	100
1	833k	125k	561k	65k	384k	117k	287k	44k	383k	65k	297k	26k
2	749k	124k	481k	64k	448k	122k	334k	48k	380k	101k	249k	39k
3	826k	128k	519k	64k	691k	181k	449k	65k	343k	150k	244k	53k
4	764k	124k	473k	64k	591k	201k	387k	71k	325k	135k	217k	57k
5	773k	125k	461k	64k	463k	202k	313k	74k	322k	150k	207k	56k
10	781k	125k	486k	63k	506k	201k	329k	80k	321k	159k	195k	57k
15	833k	126k	522k	64k	599k	216k	372k	85k	325k	158k	193k	57k
20	815k	125k	476k	63k	625k	212k	368k	89k	315k	156k	174k	58k
25	763k	124k	469k	63k	638k	211k	373k	92k	301k	156k	173k	56k
31	793k	126k	470k	63k	627k	212k	356k	94k	316k	156k	165k	59k

In our experimental set-up, we let a number of mutators repeatedly create a tree of nodes, read it a number of times and release it again. One process is the primary garbage collector process. If a mutator fails a number of times to obtain a new node, it will first try to yield the processor to assign more processor capacity to the collector because this turns out to be most efficient. But in order to guarantee the lock free nature of the algorithm, this mutator must eventually join the garbage collection process, if it fails to obtain free nodes continuously.

More concretely, in the experiments reported in Table 1, we use P mutators and memory consists of a small array of $512*P$ nodes. We let each mutator create a large ($>10^5$) number of nodes. Each mutator that must obtain a node tries 15 times to find a free node before yielding its processor. The trees generated consist either of a single node or of 255 nodes. The table provides the number of nodes that could be created and read per second (by all processors simultaneously). The letter 'k' indicates that the numbers refer to thousands of nodes.

In the columns marked with 0 these nodes are read 0 times, and in the columns marked with 100 these nodes are read a 100 times by the mutator that created it. As stated above there is one additional process assigned to do garbage collection (so there are $P+1$ processes in total). The code that was used for the experiments contained some integrity checks, some optimisations and was compiled with gcc 3.4.3 using the -O2 flag.

The experiments have been carried out on a one, two and four processor machine. All machines were Intel Linux machines of the following types:

- The single processor uses a Pentium 4 cpu of 3 Ghz with 1 Mb cache.
- The two processor machine contained two Xeon CPUs of 3 Ghz with 512 kb of cache each.
- The four processor machine has four Intel Xeon CPUs of 2 Ghz with 512 kb of cache each. In this experiment we spaced some of the data out to prevent cache trashing leading to a 30% increase in performance when not reading the nodes. Doing this on the single processor machine leads to a substantial decrease in performance.

The load linked, store conditional and verify link statements have been implemented using the 64 bit compare and swap (cmpxchg8B) instruction available on Intel Pentium processors (see [46] for the implementation). This limits parallelism to 32 processes, but does not have problems with wrap around as the implementation in [45]. With a different implementation of the load linked, store conditional and verify link statements, there is no restriction to the number of processes.

The columns of the table show that there is no loss in performance when increasing the number of mutator processes. The amount of work a processor can do basically remains constant. In the case of reading often on a parallel processor machine, we see the overhead of the single garbage collector disappear and performance increase with the number of processes. In the case of four processors and building trees of 255 nodes, we see a substantial decrease in performance, which we attribute to heavy interprocessor communications. Nodes are relatively scarce in this case and move from processor to processor.

We have also experimented with concurrent collectors. Testing showed no loss or corruption of data, corroborating the correctness of the algorithm. Performance degraded in this case, because we did not implement an efficient distribution of the work among the collectors.

Summarizing, the experiment performed must be regarded as merely a proof of concept. The question of optimizing the code and getting it up to speed is beyond the scope of this paper. It could e.g. be advisable to use local lists of free

nodes per processor. In the present set-up, free nodes are found by "randomly" picking a node and inspecting whether it is free.

7. Conclusions

We present a lock-free parallel algorithm for mark&sweep GC in a realistic model by means of synchronization primitives *load-linked/store-conditional* (LL/SC) or CAS offered by machine architectures. Our algorithm allows one to collect a circular data structure. It makes no assumption on the maximum number of mutators and collectors that can operate concurrently during GC, although the machine architecture may limit the number of processes that can use the primitives. The efficiency of GC can be enhanced when more processors are involved in it. Providing *Send* and *Receive*, our algorithm can be adapted to a distributed system, in which all processors cooperatively traverse the entire data graph by exchanging "messages" to access remote nodes.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our correctness proof presented in the paper is not flawed, we use the higher-order interactive theorem prover PVS for mechanical support. PVS has a convenient specification language and contains a proof checker which allows users to construct proofs interactively, to automatically execute trivial proofs, and to check these proofs mechanically. At several occasions where PVS refused to let a proof be finished, we actually found a mistake and had to correct previous versions of the algorithm. For the complete mechanical proof, we refer the reader to [28].

The entrenched problem inherited from classical mark&sweep algorithms is that our algorithm may also result in severe memory fragmentation, with lots of small blocks. It is possible that there will be no block of memory on the free list large enough to hold a large object, such as an array. Thus, it is important to move free blocks that happen to be adjacent in memory. We plan in the future to incorporate some appropriate copying technique in our algorithm.

In our opinion, we found a complicated garbage collection algorithm that has been proven correct by a complicated analysis. It would be preferable to be able to present a systematic path from the specification to the implementation. Indeed, in order to proceed in this work of designing provably correct lock-free algorithms, we need to improve the design process, probably by a more integrated refinement approach.

Acknowledgements

We gratefully acknowledge the positive and critical feedback of several referees.

Appendix A. Invariants

We present here all invariants and lemmas whose validity has been verified with the theorem prover PVS. In the invariants and the lemmas below, we use the relations $R(x)$, $R(p,x)$, $R1(p,x)$, $\triangleright_q$ defined in Sections 2, 2, 3.1 and 4.2, respectively. The relation $\xrightarrow{*}$ is defined in Section 2. The relation $\xrightarrow{M*}$ is the reflexive transitive closure of relation $\xrightarrow{M}$ on nodes defined by:

$$z \xrightarrow{M} x \equiv (black(z) \wedge \neg inGC[z] \wedge \exists k \in [1 \ldots \texttt{arity}[z]] : \texttt{child}[z,\ k] = x) \\ \vee\ (grey(z) \wedge \exists k \in [1 \ldots \texttt{ari}[z]] : \texttt{child}[z,\ k] = x).$$

We define the jth ancestor of a node x by the recursive function:

$$anc(x,\ j) \equiv ((j = 0\ \vee\ \texttt{father}[x] \leq 0)\ ?\ x\ :\ anc(\texttt{father}[x],\ j-1)).$$

Main invariants:

I1: $white(x)\ \Rightarrow\ \neg R(x)$

I2: $white(x)\ \equiv\ x \in \texttt{free}$

I3: $554 \leq pc_p \leq 559\ \Rightarrow\ x_p \in roots_p$

I4: $grey(x)\ \Rightarrow\ \exists\ p : md_p = \texttt{shRnd}$

I5: $\texttt{srcnt}[x] - \texttt{freecnt}[x] = \sharp(\{p \mid x \in roots_p\}) + \sharp(\{(p,\ q) \mid (\texttt{Mbox}[p,\ q] = x \wedge \neg(pc_q = 559 \wedge p = r_q)) \vee (pc_p = 508 \wedge x_p = x \wedge q = r_p)\})$.

Invariants about the stability of the preconditions in the offered procedures:

I6: $250 \le pc_p \le 258 \Rightarrow R(p, x_p) \wedge R(p, y_p)$

I7: $pc_p = 280 \Rightarrow R(p, x_p)$

I8: $300 \le pc_p \le 308 \vee (100 \le pc_p \le 180 \wedge return_p = 300) \wedge 1 \le k \le n_p \Rightarrow R(p, c_p[k])$

I9: $pc_p = 400 \vee 500 \le pc_p \le 508 \Rightarrow R(p, x_p)$

I10: $500 \le pc_p \le 508 \Rightarrow \mathtt{Mbox}[p, r_p] = 0$

I11: $550 \le pc_p \le 559 \Rightarrow \mathtt{Mbox}[r_p, p] \neq 0$

Invariants that apply globally:

I12: $rnd_p \le \mathtt{shRnd}$

I13: $\mathtt{shRnd} \le \mathtt{round}[x] \le \mathtt{shRnd} + 1$

I14: $inGC[x] \Rightarrow \mathtt{round}[x] = \mathtt{shRnd} + 1$

I15: $\mathtt{shRnd} < \mathtt{round}[x] \Rightarrow \exists\, p : rnd_p = \mathtt{shRnd}$

I16: $grey(x) \Rightarrow \mathtt{shRnd} < \mathtt{round}[x]$

I17: $grey(x) \Rightarrow inGC[x]$

I18: $white(x) \Rightarrow \neg R1(x)$

I19: $white(x) \Rightarrow \mathtt{father}[x] \le -1$

I20: $grey(x) \vee \mathtt{father}[x] \ge 0 \Rightarrow \mathtt{ari}[x] \le \mathtt{arity}[x]$

I21: $grey(x) \wedge \mathtt{father}[x] > 0 \Rightarrow \exists\, k \in [1 \ldots \mathtt{ari}[\mathtt{father}[x]]] : \mathtt{child}[\mathtt{father}[x], k] = x$

I22: $x \neq \mathtt{father}[x]$

I23: $\mathtt{father}[x] > 0 \wedge j > 0 \Rightarrow anc(x, j) \neq x$

I24: $\mathtt{father}[x] = 0 \wedge grey(x) \Rightarrow \mathtt{srcnt}[x] > 0$

I25: $(\exists\, p : x \in roots_p) \Rightarrow \mathtt{srcnt}[x] - \mathtt{freecnt}[x] > 0$

I26: $\neg R(x) \Rightarrow \mathtt{srcnt}[x] - \mathtt{freecnt}[x] = 0$

I27: $R1(x) \wedge (grey(x) \vee (black(x) \wedge \neg inGC[x]))$
$\Rightarrow \exists\, w : \mathtt{srcnt}[w] > 0 \wedge (\mathtt{father}[w] = 0 \vee \neg inGC[w]) \wedge w \xrightarrow{M*} x$

I28: $return_p = 200 \vee return_p = 300 \vee return_p = 450$

Invariants about the first phase of GC:

I29: $101 \le pc_p \le 110 \wedge rnd_p = \mathtt{shRnd} \wedge (x \notin toBeC_p \vee inGC[x])$
$\Rightarrow \mathtt{round}[x] = rnd_p + 1$

I30: $101 \le pc_p \le 110 \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{father}[x] = 0 \wedge inGC[x] \wedge black(x)$
$\Rightarrow \exists\, r : \neg(101 \le pc_r \le 110) \wedge rnd_r = \mathtt{shRnd}$

I31: $101 \le pc_p \le 110 \wedge rnd_p = \mathtt{shRnd} \wedge \neg inGC[x]$
$\wedge (x \notin toBeC_p \vee \mathtt{round}[x] = rnd_p + 1)$
$\Rightarrow \exists\, r : \neg(101 \le pc_r \le 110) \wedge rnd_r = \mathtt{shRnd}$

I32: $101 \le pc_p \le 110 \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{father}[x] > 0$
$\wedge (x \notin toBeC_p \vee inGC[x])$
$\Rightarrow \exists\, r : \neg(101 \le pc_r \le 110) \wedge rnd_r = \mathtt{shRnd}$

I33: $101 \le pc_p \le 110 \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{father}[x] \ge 0$
$\wedge (x \notin toBeC_p \vee \mathtt{round}[x] = rnd_p + 1) \wedge \mathtt{srcnt}[x] = 0$
$\Rightarrow \exists\, r : \neg(101 \le pc_r \le 110) \wedge rnd_r = \mathtt{shRnd}$

Invariants about the second phase of GC:

I34: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \Rightarrow \mathtt{round}[x] = rnd_p + 1$

I35: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \Rightarrow inGC[x]$

I36: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{father}[w] \le -1 \Rightarrow \neg(\exists\, x : \mathtt{father}[x] = w)$

I37: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \wedge grey(x) \wedge \mathtt{father}[x] > 0$
$\Rightarrow grey(\mathtt{father}[x]) \wedge \mathtt{father}[\mathtt{father}[x]] \ge 0$

I38: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{father}[x] > 0 \wedge grey(x)$
$\Rightarrow \exists\, j \in \mathtt{Index} : \mathtt{father}[anc(x, j)] = 0 \wedge grey(anc(x, j))$

I39: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \wedge \neg R1(x) \wedge grey(x)$
$\Rightarrow \mathtt{father}[x] = -1$

I40: $\neg(101 \le pc_p \le 110) \wedge rnd_p = \mathtt{shRnd} \wedge black(w) \wedge \mathtt{father}[w] \ge 0$
$\Rightarrow \forall\, k \in [1 \ldots \mathtt{ari}[w]] : (\mathtt{father}[\mathtt{child}[w, k]] = w \vee \mathtt{father}[\mathtt{child}[w, k]] < 0$
$\Rightarrow black(\mathtt{child}[w, k]))$

I41: $pc_p = 121 \Rightarrow rnd_p \neq \mathtt{shRnd} \vee toBeC_p = \emptyset$

I42: $\neg(101 \le pc_p \le 121) \wedge rnd_p = \mathtt{shRnd} \wedge \mathtt{srcnt}[x] > 0 \wedge \mathtt{father}[x] = 0$
$\wedge \neg(x \in toBeD_p \vee (150 \le pc_p \le 180 \wedge x = x_p))$
$\Rightarrow black(x)$

I43: $(122 \le pc_p \le 127 \vee 150 \le pc_p \le 180) \wedge rmd_p = \texttt{shRnd} \wedge x \notin toBeC_p$
$\Rightarrow \texttt{father}[x] \ge 0$

I44: $((122 \le pc_p \le 127 \vee 150 \le pc_p \le 180) \wedge rmd_p = \texttt{shRnd} \wedge x \in toBeD_p$
$\wedge \texttt{father}[x] = 0) \vee (pc_p = 150 \wedge rmd_p = \texttt{shRnd} \wedge x = x_p)$
$\Rightarrow x \in toBeC_p$

I45: $(122 \le pc_p \le 127 \vee 150 \le pc_p \le 180) \wedge rmd_p = \texttt{shRnd} \wedge x \notin toBeC_p$
$\Rightarrow \neg(\exists w : \texttt{father}[x] = w \wedge (w \in set_p \vee w \in toBeC_p))$

I46: $122 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd} \wedge grey(x)$
$\Rightarrow x \in toBeC_p \vee (pc_p \ge 151 \wedge (x \in set_p \vee \exists i \in [1 \ldots head_p] : x = stack_p[i]))$

Invariants about procedure Mark_stack:

I47: $pc_p = 150 \Rightarrow x_p \notin toBeD_p$

I48: $150 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd}$
$\Rightarrow (grey(x_p) \wedge \texttt{father}[x_p] = 0 \wedge \texttt{srcnt}[x_p] > 0)$
$\vee\ (black(x_p) \wedge \texttt{father}[x_p] = 0)$

I49: $151 \le pc_p \le 180 \wedge x \in toBeC_p$
$\Rightarrow \neg(x \in set_p \vee \exists i \in [1 \ldots head_p] : x = stack_p[i])$

I50: $151 \le pc_p \le 180 \wedge (\exists i \in [1 \ldots head_p] : x = stack_p[i])$
$\Rightarrow x \notin set_p \wedge x \notin toBeC_p$

I51: $151 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd}$
$\Rightarrow x_p \in set_p \vee black(x_p) \vee (\exists i \in [1 \ldots head_p] : x_p = stack_p[i])$

I52: $151 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd} \wedge (x \in set_p \vee \exists i \in [1 \ldots head_p] : x = stack_p[i])$
$\Rightarrow \neg white(x) \wedge \texttt{father}[x] \ge 0$

I53: $151 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd} \wedge grey(x)$
$\wedge (x \in set_p \vee \exists i \in [1 \ldots head_p] : x = stack_p[i])$
$\Rightarrow R1(x) \wedge \texttt{father}[x] \ge 0$

I54: $151 \le pc_p \le 180 \wedge rmd_p = \texttt{shRnd} \wedge (\exists i \in [1 \ldots head_p] : w = stack_p[i])$
$\Rightarrow \forall k \in [1 \ldots \texttt{ari}[w]] : (\texttt{father}[\texttt{child}[w,\ k]] \ge 0 \vee black(\texttt{child}[w,\ k])$
$\wedge (\texttt{father}[\texttt{child}[w,\ k]] = w \Rightarrow \texttt{child}[w,\ k] \in set_p \vee black(\texttt{child}[w,\ k])$
$\vee (\exists j \in [1 \ldots head_p] : \texttt{child}[w,\ k] = stack_p[j]) \wedge (\forall m,\ n \in [1 \ldots head_p] :$
$w = stack_p[m] \wedge x = stack_p[n] \Rightarrow m < n))$
$\vee (158 \le pc_p \le 164 \wedge w_p = w \wedge k \ge j_p)$

I55: $pc_p = 158 \wedge rmd_p = \texttt{shRnd} \Rightarrow j_p = 1 \vee 1 < j_p \le \texttt{ari}[w_p] + 1$

I56: $158 \le pc_p \le 164 \wedge rmd_p = \texttt{shRnd}$
$\Rightarrow k_p = \texttt{ari}[w_p] \wedge \forall j \in [1 \ldots k_p] : ch_p[j] = \texttt{child}[w_p,\ j]$

I57: $158 \le pc_p \le 164 \wedge rmd_p = \texttt{shRnd} \wedge \neg(x \in toBeC_p \vee \exists j \in [1 \ldots j_p - 1] : x = \texttt{child}[w_p,\ j])$
$\Rightarrow \texttt{father}[x] \ne w_p$

I58: $158 \le pc_p \le 164 \wedge rmd_p = \texttt{shRnd}$
$\Rightarrow \forall k \in [1 \ldots j_p - 1] : (grey(\texttt{child}[w_p,\ k]) \Rightarrow \texttt{father}[\texttt{child}[w_p,\ k]] \ge 0$
$\wedge (\texttt{father}[\texttt{child}[w_p,\ k]] = w_p \Rightarrow \texttt{child}[w_p,\ k] \in set_p))$

I59: $158 \le pc_p \le 165 \Rightarrow \exists i \in [1 \ldots head_p] : w_p = stack_p[i]$

I60: $159 \le pc_p \le 164 \wedge rmd_p = \texttt{shRnd} \Rightarrow 1 \le j_p \le \texttt{ari}[w_p] \wedge y_p = \texttt{child}[w_p,\ j_p]$

I61: $168 \le pc_p \le 180 \Rightarrow rmd_p \ne \texttt{shRnd} \vee set_p = \emptyset$

I62: $170 \le pc_p \le 176 \Rightarrow head_p \ne 0$

I63: $pc_p = 180 \Rightarrow rmd_p \ne \texttt{shRnd} \vee head_p = 0$

Invariants about the third phase of GC:

I64: $129 \le pc_p \le 137 \wedge rmd_p = \texttt{shRnd} \wedge grey(x) \Rightarrow \neg R1(x)$

I65: $pc_p = 134 \wedge \texttt{round}[x] = rmd_p + 1 \wedge grey(x) \Rightarrow \neg R(x) \wedge x \notin \texttt{free}$

I66: $rmd_p = \texttt{shRnd} \wedge grey(x) \Rightarrow 101 \le pc_p \le 134 \vee 150 \le pc_p \le 180$

I67: $pc_p = 135 \Rightarrow rmd_p \ne \texttt{shRnd} \vee toBeC_p = \emptyset$

I68: $rmd_p = \texttt{shRnd} \Rightarrow 101 \le pc_p \le 135 \vee 150 \le pc_p \le 180$

Invariants outside GC:

I69: $pc_p = 450 \vee (100 \le pc_p \le 180 \wedge return_p = 450) \Rightarrow R(p,\ z_p)$

I70: $500 \le pc_p \le 508 \Rightarrow \texttt{Mbox}[p,\ r_p] = 0$

I71: $552 \le pc_p \le 559 \Rightarrow x_p = \texttt{Mbox}[r_p,\ p] \wedge x_p \ne 0$

I72: $pc_p = 558 \Rightarrow \texttt{srcnt}[x_p] > 1$

Main lemmas:
V1: $R(p,\ x) \wedge I18 \wedge I25 \wedge p \neq q \ \ \triangleright_q \ \ R(p,\ x)$
V2: $\neg white(x) \wedge \neg R1(x) \wedge I6 \wedge I8 \wedge I9 \wedge I25 \wedge I63 \wedge I66 \ \triangleright \ \neg R1(x)$

Appendix B. Dependencies between invariants

Let us write "ϕ **from** $\psi_1, \ldots, \psi_n$" to denote that ϕ can be proved to be an invariant using that $\psi_1, \ldots, \psi_n$ hold in the precondition of every step. We write "$\phi \ \Leftarrow \ \psi_1, \ldots, \psi_n$" to denote that ϕ can be directly derived from $\psi_1, \ldots, \psi_n$. We have verified the following "**from**" and "$\Leftarrow$" relations mechanically:

I1 $\Leftarrow$ I3, I5, I18, I71
I2 **from** : *true*
I3 **from** : I28
I4 $\Leftarrow$ I12, I15, I16
I5 **from** : I18, I25, I28, I70, I71
I6 **from** : I18, I25, I28
I7 **from** : I18, I25, I28
I8 **from** : I18, I25, I28
I9 **from** : I18, I25, I28
I10 **from** : I28
I11 **from** : I28
I12 **from** : *true*
I13 **from** : I12, I34
I14 **from** : I12, I13
I15 **from** : I12, I13, I34
I16 **from** : I12, I13, I66
I17 **from** : I66
I18 **from** : I6, I8, I9, I12, I16, I25, I64, I69
I19 **from** : I12, I16, I18, I39, I64
I20 **from** : I19
I21 **from** : I12, I13, I14, I15, I17, I20, I32, I34, I37, I60, I66
I22 $\Leftarrow$ I23
I23 **from** : I13, I16, I36, I50, I59
I24 **from** : *true*
I25 $\Leftarrow$ I5
I26 $\Leftarrow$ I3, I5, I9
I27 **from** : I6, I8, I9, I12, I13, I14, I16, I17, I18, I20, I21, I24, I25, I35, I37, I38, I54, I61, I64, I66, I69
I28 **from** : *true*
I29 **from** : I12, I13, I14, I28
I30 **from** : I12, I13, I14, I15, I16, I19, I28, I68
I31 **from** : I12, I13, I15, I28, I35, I68
I32 **from** : I12, I13, I14, I15, I28, I29, I31, I34, I68
I33 **from** : I12, I13, I15, I28, I29, I34, I68
I34 **from** : I12, I13, I29, I34
I35 **from** : I12, I31, I34
I36 **from** : I12, I32, I34, I52, I59
I37 **from** : I12, I21, I28, I32, I34, I39, I40, I52, I54, I59, I60, I61, I64
I38 $\Leftarrow$ I23, I32, I35, I37
I39 **from** : I12, I18, I19, I20, I21, I24, I27, I28, I33, I34, I35, I37, I38, I40, I52, I53, I54, I59, I60, I61
I40 **from** : I12, I19, I20, I28, I30, I31, I32, I34, I35, I54, I61, I62
I41 **from** : I12, I28
I42 **from** : I9, I12, I18, I24, I25, I34, I42, I51, I61, I63
I43 **from** : I12, I28, I34, I43, I48
I44 **from** : I12, I28, I34, I35, I47
I45 **from** : I12, I28, I43, I44, I48, I50, I59
I46 **from** : I12, I34, I61, I63
I47 **from** : I28
I48 **from** : I12, I18, I19, I24, I28, I34, I39, I64
I49 **from** : I28
I50 **from** : I28, I49
I51 **from** : I12, I28, I34, I52

I52 **from** : I12, I28, I34, I48, I53, I64
I53 **from** : I12, I18, I19, I20, I21, I24, I27, I28, I34, I35, I37, I38, I40, I48, I52, I54, I59, I60, I61
I54 **from** : I12, I18, I19, I20, I22, I28, I34, I35, I40, I43, I50, I52, I53, I55, I56, I57, I58, I60, I62
I55 **from** : I12, I28, I34, I56, I60
I56 **from** : I12, I20, I28, I34, I52, I59
I57 **from** : I12, I19, I20, I28, I34, I43, I45, I52, I55, I59, I60
I58 **from** : I12, I20, I28, I34, I35, I43, I52, I55, I56, I57, I59, I60
I59 **from** : I28
I60 **from** : I12, I20, I28, I34, I52, I56, I59
I61 **from** : I12, I28
I62 **from** : I28
I63 **from** : I12, I28
I64 **from** : I6, I8, I9, I12, I25, I27, I34, I35, I42
I65 ⇐ I2, I3, I5, I12, I16, I64, I71
I66 **from** : I12, I34, I46
I67 **from** : I12, I28
I68 **from** : I12
I69 **from** : I18, I25
I70 **from** : I28
I71 **from** : I28, I70
I72 ⇐ I3, I5, I71

Appendix C. The low-level lock-free algorithm

C.1. Data structure

Constants
N, C as in Fig. 1
P is the total numer of processes
Types
`colorType`, `nodeType` as in Fig. 3
`Index`, `RefIndex` as in Fig. 11
Shared variables
`node` : **array** [`RefIndex`] **of** `nodeType`;
`indir` : **array** [`Index`] **of** `RefIndex`;
`Mbox` : **array** [`Process`, `Process`] **of** `Index` ∪ {0};
`shRnd` : ℕ;
Private variables
roots : **subset of** `Index`; % a set of root nodes
rnd : ℕ; % private copy of "shRnd", initially 0!
toBeC : **subset of** `Index`; % a set of nodes to be checked
mp : `RefIndex`; % the pointer to the private copy of a node

Initialization:
`shRnd` = 1 ∧ ∀ x ∈ `Index` : (`indir`[x] = x ∧ `round`[`indir`[x]] = 1);
∀ p ∈ `Process` : $mp_p = N + p$;
all other variables are equal to the minimal values in their respective domains.

C.2. Algorithm

proc *GCollect*() =
local *m* : `RefIndex`; *x* : `Index`; *toBeD* : **subset of** `Index`;
% first phase
100: *rnd* := `shRnd`; *toBeC* := `Index`;
101: **while** `shRnd` = *rnd* ∧ *toBeC* ≠ ∅ **do**
choose *x* ∈ *toBeC*;
while *true* **do**
102: *m* := *LL*(`indir`[*x*]);
103: `node`[*mp*] := `node`[*m*];
104: **if** `round`[*mp*] = *rnd* **then**

```
105:            round[mp] := rnd + 1; ari[mp] := arity[mp];
                if color[mp] = black then color[mp] := grey; fi;
                if srcnt[mp] > 0 then father[mp] := 0; else father[mp] := −1; fi;
106:            if SC(indir[x], mp) then toBeC := toBeC − {x}; mp := m; break; fi;
107:          elseif VL(indir[x]) then toBeC := toBeC − {x}; break; fi;
        od;
      od;
% second phase
110:  toBeC := Index ; toBeD := Index;
111:  while shRnd = rnd ∧ toBeD ≠ ∅ do
        choose x ∈ toBeD;
        while true do
112:      m := LL(indir[x]);
113:      node[mp] := node[m];
114:      if father[mp] = 0 then
116:        if VL(indir[x]) then
            toBeD := toBeD − {x};
            Mark_stack(x); break; fi;
117:      elseif VL(indir[x]) then toBeD := toBeD − {x}; break; fi;
        od;
      od;
% last phase
120:  while shRnd = rnd ∧ toBeC ≠ ∅ do
        choose x ∈ toBeC;
        while true do
121:      m := LL(indir[x]);
122:      node[mp] := node[m];
123:      if round[mp] = rnd + 1 ∧ color[mp] = grey then
124:        color[mp] := white;
125:        if SC(indir[x], mp) then toBeC := toBeC − {x}; mp := m; break; fi;
126:      elseifVL(indir[x]) then toBeC := toBeC − {x}; break; fi;
        od;
      od;
127:  CAS(shRnd, rnd, rnd + 1);
128:  return;
end GCollect.

proc  Mark_stack(x : Index) =
      local w, y : Index; suc : Bool; j, k : ℕ;
      stack : Stack; head : ℕ; set : subset of Index;
      ch : [1 . . . C] of Index; m, n : RefIndex;
150:  toBeC := toBeC − {x}; set := {x}; head := 0;
151:  while shRnd = rnd ∧ set ≠ ∅ do
        choose w ∈ set;
        while true do
152:      m := LL(indir[w]);
153:      node[mp] := node[m];
154:      if color[mp] = grey ∧ round[mp] = rnd + 1 then
155:        k := ari[mp];
          for j := 1 to k do ch[j] := child[mp, j]; od;
156:        if VL(indir[w]) then
            set := set − {w}; head++; stack[head] := w; j := 1;
157:        while shRnd = rnd ∧ j ≤ k do
              y := ch[j];
              if y ∈ toBeC then
                while true do
158:              n := LL(indir[y]);
159:              node[mp] := node[n];
160:              if (father[mp] = −1 ∨ father[mp] = w)
                          ∧round[mp] = rnd + 1 then
161:                if father[mp] = −1 then father[mp] := w; fi;
162:                if SC(indir[y], mp) then
```

```
                        toBeC := toBeC − {y}; mp := n;
                        set := set + {y}; break; fi;
163:                elseif VL(indir[y]) then break; fi;
                od; fi;
              j := j + 1;
            od;
            break; fi;
164:      elseif VL(indir[w]) then set := set − {w}; break; fi;
        od;
    od;
170: while shRnd = rnd ∧ head ≠ 0 do
        y := stack[head];
        while true do
171:      m := LL(indir[y]);
172:      node[mp] := node[m];
173:      if color[mp] = grey ∧ round[mp] = rnd + 1 then
174:        color[mp] := black;
          srcnt[mp] := srcnt[mp] − freecnt[mp]; freecnt[mp] := 0;
175:        if SC(indir[y], mp) then mp := m; head--; break; fi;
176:      elseifVL(indir[y]) then head--; break; fi;
        od;
    od;
180: return;
end Mark_stack.

proc Create() : Index =
    local m : RefIndex; x : Index;
    while true do
200:   choose x ∈ Index;
201:   m := LL(indir[x]);
202:   node[mp] = node[m];
203:   if color[mp] = white then
204:     color[mp] := black; srcnt[mp] := 1; arity[mp] := 0;
205:     if SC(indir[x], mp) then
           roots := roots + {x};
           mp := m; break; fi;
206:   elseif time to do GC then
           GCollect(); fi;
    od;
207: return x
end Create.

proc AddChild(x, y : Index) : Bool =
{R(self, x) ∧ R(self, y)}
    local m : RefIndex; suc : Bool;
250: suc := false;
    while true do
251:   m := LL(indir[x]);
252:   node[mp] := node[m];
253:   if arity[mp] < C then
254:     arity[mp]++;
       child[mp, arity[mp]] := y;
255:     if SC(indir[x], mp) then
           mp := m; suc := true; break; fi;
256:   elseif VL(indir[x]) then break; fi;
    od;
257: return suc
end AddChild.

proc GetChild(x : Index, rth : Index) : 0 . . . N =
{R(self, x)}
    local m : RefIndex; y : Index;
```

```
      while true do
280:    m := LL(indir[x]);
281:    node[mp] := node[m];
282:    if 1 ≤ rth ≤ arity[mp] then y := child[mp, rth]; else y := 0; fi;
283:    if VL(indir[x]) then break; fi;
      od;
284:  return y
end GetChild.

proc Make(c : array [1...C] of Index, n : 1...C) : Index =
{∀ j : 1...n : R(self, c[j])}
      local m : RefIndex; x : Index; j : ℕ;
      while true do
300:    choose x ∈ Index;
301:    m := LL(indir(x));
302:    node[mp] := node[m];
303:    if color[mp] = white then
304:      color[mp] := black;
          srcnt[mp] := 1; arity[mp] := n;
          for j := 1 to n do child[mp, j] := c[j] od;
305:      if SC(indir(x), mp) then
            roots := roots + {x};
            mp := m; break; fi;
306:    elseif time to do GC then
          GCollect(); fi;
      od;
307:  return x
end Make.

proc Protect(x : Index) =
{R(self, x) ∧ x ∉ roots}
      local m : RefIndex;
      while true do
400:    m := LL(indir[x]);
401:    node[mp] := node[m];
402:    srcnt[mp]++;
403:    if SC(indir[x], mp) then
          roots := roots + {x};
          mp := m; break; fi;
      od;
404:  return
end Protect.

proc UnProtect(z : Index) =
{z ∈ roots}
      local m : RefIndex;
      while true do
450:    m := LL(indir[z]);
451:    node[mp] := node[m];
452:    freecnt[mp]++;
453:    if SC(indir[x], mp) then
          roots := roots \ {z};
          mp := m; break; fi;
      od;
454:  return
end UnProtect.

proc Send(x : Index, r : Process) =
{R(self, x) ∧ Mbox[self, r] = 0}
      local m : RefIndex;
      while true do
500:    m := LL(indir[x]);
```

```
501:      node[mp] := node[m];
502:      srcnt[mp]++;
503:      if SC(indir[x], mp) then
            mp := m;
504:        Mbox[self, r] := x; break; fi;
        od;
505:  return
end Send.

proc Receive(r : Process) : 0 ... N =
{Mbox[r, self] ≠ 0}
        local x : Index;
550:    x := Mbox[r, self];
551:    if x ∉ roots then
          roots := roots ∪ {x};
          Mbox[r, self] := 0;
        else
          while true do
552:        m := LL(indir[x]);
553:        node[mp] := node[m];
554:        srcnt[mp]--;
555:        if SC(indir[x], mp) then
              mp := m;
559:          Mbox[r, self] := 0;
              break; fi;
          od; fi;
560:    return
end Receive.

proc Check(r, q : Process) : Bool
        local suc : Bool;
600:    suc := (Mbox[r, q] = 0);
601:    return suc;
end Check.
```

References

[1] K.R. Apt, E.-R. Olderog, Verification of Sequential and Concurrent Programs, Springer, New York, 1991.

[2] H. Azatchi, Y. Levanoni, H. Paz, E. Petrank, An on-the-fly mark and sweep garbage collector based on sliding views, in: Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, ACM Press, 2003, pp. 269–281.

[3] R.J.R. Back, J. von Wright, Stepwise refinement of distributed systems: Models, formalism, correctness: Refinement calculus, in: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems, in: Lecture Notes in Computer Science, vol. 430, Springer-Verlag, 1990, pp. 42–93.

[4] G. Barnes, A method for implementing lock-free data structures, in: Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures, June 1993, pp. 261–270.

[5] M. Ben-Ari, Algorithms for on-the-fly garbage collection, ACM Transactions on Programming Languages and Systems 6 (3) (1984) 333–344.

[6] B.N. Bershad, Practical considerations for non-blocking concurrent objects, in: Proceedings of the Thirteenth International Conference on Distributed Computing Systems, 1993, pp. 264–274.

[7] H. Boehm, A.J. Demers, S. Shenker, Mostly parallel garbage collection, in: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, ACM Press, 1991, pp. 157–164.

[8] M.G.J. van den Brand, H.A. de Jong, P. Klint, P.A. Olivier, Efficient annotated terms, Software, Practice and Experience 30 (3) (2000) 259–291.

[9] K.M. Chandy, Parallel Program Design: A Foundation, Addison-Wesley Longman Publishing Co., Inc., 1988.

[10] E. Cohen, L. Lamport, Reduction in TLA, in: Proceedings of the 9th International Conference on Concurrency Theory, 1998, pp. 317–331.

[11] C. Cornes, J. Courant, et al., The coq proof assistant — reference manual v 6.1, 1997.

[12] D.L. Detlefs, P.A. Martin, M. Moir, G.L. Steele Jr., Lock-free reference counting, Distributed Computing 15 (4) (2002) 255–271.

[13] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens, On-the-fly garbage collection: An exercise in cooperation, Communications of the ACM 21 (11) (1978) 966–975.

[14] D. Doligez, X. Leroy, A concurrent generational garbage collector for a multi-threaded implementation of ML, in: Proceedings of the 1993 ACM Symposium on Principles of Programming Languages, January 1993, pp. 113–123.

[15] T. Endo, K. Taura, A. Yonezawa, A scalable mark–sweep garbage collector on large-scale shared-memory machines, in: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing, (CDROM), ACM Press, 1997, pp. 1–14.

[16] C. Flood, D. Detlefs, N. Shavit, C. Zhang, Parallel garbage collection for shared memory multiprocessors, in: Usenix Java Virtual Machine Research and Technology Symposium, JVM '01, Monterey, CA, April 2001.
[17] H. Gao, J.F. Groote, W.H. Hesselink, Lock-free dynamic hash tables with open addressing, Distributed Computing 17 (2005) 21–42.
[18] H. Gao, J.F. Groote, W.H. Hesselink, Almost wait-free resizable hashtables (extended abstract), in: Proceedings of 18th International Parallel and Distributed Processing Symposium, IPDPS, IEEE Computer Society, April 2004.
[19] H. Gao, J.F. Groote, W.H. Hesselink, Lock-free parallel garbage collection by mark&sweep, Technical Report CS-Report 04-31, Eindhoven University of Technology, The Netherlands, 2004.
[20] H. Gao, J.F. Groote, W.H. Hesselink, Lock-free parallel garbage collection, in: Y. Pan, D. Chen, M. Guo, J. Cao, J. Dongarra (Eds.), Proceedings of Third International Symposium on Parallel and Distributed Processing and Applications, ISPA'05, in: LNCS, vol. 3758, Springer, 2005, pp. 263–274.
[21] H. Gao, W.H. Hesselink, A formal reduction for lock-free parallel algorithms, in: Proceedings of the 16th Conference on Computer Aided Verification, CAV, July 2004.
[22] H. Gao, W.H. Hesselink, A general lock-free algorithm using compare-and-swap, Information and Computation, in press (doi:10.1016/j.ic.2006.10.003). http://www.cs.rug.nl/~wim/mechver/lockfree_reduction, 2004.
[23] J.F. Groote, W.H. Hesselink, S. Mauw, R. Vermeulen, An algorithm for the asynchronous write-all problem based on process collision, Distributed Computing 14 (2001) 75–81.
[24] K. Havelund, Mechanical verification of a garbage collector, in: J. Rolim, et al. (Eds.), Parallel and Distributed Processing (Combined Proceedings of 11 Workshops), in: Lecture Notes in Computer Science, vol. 1586, Springer-Verlag, April 1999, pp. 1258–1283. Presented at the Workshop on Formal Methods for Parallel Programming: Theory and Applications, FMPPTA.
[25] M. Herlihy, A methodology for implementing highly concurrent data objects, ACM Transactions on Programming Languages and Systems 15 (5) (1993) 745–770.
[26] M.P. Herlihy, V. Luchangco, M. Moir, The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure, in: Proceedings of 16th International Symposium on Distributed Computing, Springer-Verlag, October 2002, pp. 339–353.
[27] M.P. Herlihy, J.E.B. Moss, Lock-free garbage collection for multiprocessors, IEEE Transactions on Parallel and Distributed Systems 3 (3) (1992) 304–311.
[28] W.H. Hesselink, http://www.cs.rug.nl/~wim/mechver/garbage_collection.
[29] W.H. Hesselink, Using eternity variables to specify and prove a serializable database interface, Science of Computer Programming 51 (2004) 47–85.
[30] W.H. Hesselink, J.F. Groote, Wait-free concurrent memory management by Create, and Read until Deletion, Distributed Computing 14 (1) (2001) 31–39.
[31] R.L. Hudson, J.E.B. Moss, Sapphire: Copying GC without stopping the world, in: ISCOPE Conference on ACM 2001 Java Grande, ACM Press, 2001, pp. 48–57.
[32] L. Huelsbergen, J.R. Larus, A concurrent copying garbage collector for languages that distinguish (im)mutable data, in: Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press, 1993, pp. 73–82.
[33] P.B. Jackson, Verifying a garbage collection algorithm, in: Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics TPHOLs'98, in: LNCS, vol. 1479, 1998, pp. 225–244.
[34] E.H. Jensen, G.W. Hagensen, J.M. Broughton, A new approach to exclusive data access in shared memory multiprocessors, Technical Report UCRL-97663, Lawrence Livemore National Laboratory, January 1987.
[35] R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons, Inc., 1996.
[36] J.E. Jonker, On-the-fly garbage collection for several mutators, Distributed Computing 5 (1992) 187–199.
[37] P.C. Kanellakis, A.A. Shvartsman, Fault-Tolerant Parallel Computation, Kluwer Academic Publishers, 1997.
[38] L. Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 872–923.
[39] Y. Levanoni, E. Petrank, An on-the-fly reference counting garbage collector for Java, in: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, 2001, pp. 367–380.
[40] V. Luchangco, M. Moir, N. Shavit, Nonblocking k-compare-single-swap, in: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, 2003, pp. 314–323.
[41] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, 1996.
[42] H. Massalin, C. Pu, A lock-free multiprocessor OS kernel, Technical Report CUCS-005-91, Columbia University, 1991.
[43] M.M. Michael, Safe memory reclamation for dynamic lock-free objects using atomic reads and writes, in: Proceedings of the 21st Annual Symposium on Principles of Distributed Computing, ACM Press, 2002, pp. 21–30.
[44] M.M. Michael, High performance dynamic lock-free hash tables and list-based sets, in: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press, 2002, pp. 73–82.
[45] M. Moir, Practical implementations of non-blocking synchronization primitives, in: Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 1997, pp. 219–228.
[46] A.J. Mooij, Non-blocking implementations of LL, VL and SC, 2004, private communication.
[47] L. Moreau, J. Duprat, A construction of distributed reference counting, Acta Informatica 37 (8) (2001) 563–595.
[48] J. O'Toole, S. Nettles, Concurrent replicating garbage collection, in: Proceedings of the 1994 ACM Conference on LISP and Functional Programming, ACM Press, 1994, pp. 34–42.
[49] S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6 (1976) 319–340.
[50] S. Owicki, L. Lamport, Proving liveness properties of concurrent programs, ACM Transactions on Programming Languages and Systems 4 (1982) 455–495.
[51] S. Owre, N. Shankar, J.M. Rushby, D.W.J. Stringer-Calvert, PVS Version 2.4: System Guide, Prover Guide, PVS Language Reference, 2001.

[52] H. Rodrigues, R. Jones, Cyclic distributed garbage collection with group merger, in: Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98, Springer, Brussels, July 1998, pp. 249–273.
[53] D.M. Russinoff, A mechanically verified incremental garbage collector, Formal Aspects of Computing 6 (1994) 359–390.
[54] H. Sundell, P. Tsigas, Scalable and lock-free concurrent dictionaries, in: Proceedings of the 2004 ACM Symposium on Applied Computing, 2004, pp. 1438–1445.
[55] J.D. Valois, Lock-free linked lists using compare-and-swap, in: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 1995, pp. 214–222.